

Preference-based Web Service Composition

by

© *Mehdi Razeghin*

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

May 2014

St. John's

Newfoundland

Contents

List of Tables	vi
List of Figures	vii
Acknowledgements	1
1 Introduction	2
1.1 Motivation	2
1.2 Approach	4
1.3 Organization of the Thesis	6
2 Service-Oriented Architecture and Web Services	7
2.1 Software Architecture	7
2.2 Service-Oriented Architecture	8
2.2.1 Principles of SOA	9
2.2.2 Roles of Interaction in SOA	10
2.2.3 Services as the Core Component of SOA	12
2.3 Web Services	13
2.3.1 Semantics of Web Services	14

2.3.2	Web Service Composition	15
2.3.2.1	Web Service Composition via AI-planning	16
2.3.2.2	Web Service Composition via Query Rewriting	21
3	Running Example	23
3.1	Structure of the Domain	24
3.2	Available Web Services	25
3.3	Scenarios	28
4	Data Integration and Query Rewriting	30
4.1	Datalog	31
4.2	Conjunctive Queries	33
4.3	Data Integration Techniques	37
4.3.1	Global-As-View (GAV)	37
4.3.2	Local-As-View (LAV)	39
4.4	Query Rewriting Algorithms in LAV	46
4.4.1	Logical-based algorithms	46
4.4.2	Bucket-based algorithms	49
4.4.2.1	Shared-Variable Bucket algorithm	51
4.4.2.2	MiniCon algorithm	56
4.4.3	Hybrid algorithm	59
4.4.4	Graph-based Algorithm	60
4.4.5	Comparison of the Algorithms	61
4.5	Query Rewriting with Dependencies	62

5	Related Work	65
5.1	Web Service Composition	66
5.1.1	Evaluation of the Related Works	76
5.2	Query Rewriting using Views in the Presence of Dependencies	85
6	Query Rewriting in the Presence of Dependencies	87
6.1	Domain Ontology	88
6.2	Required Tools for using Domain Ontology in Query Rewriting	94
6.3	Computing Δ -contained Queries	108
6.3.1	MiniCon-FD: A Naive Query Rewriting Algorithm in the Pres-	
	ence of Full Dependencies	109
6.3.1.1	Motivation	109
6.3.1.2	The mechanism of MiniCon-FD	110
6.3.1.3	The correctness of MiniCon-FD	113
6.3.2	Optimization of MiniCon-FD	113
6.3.2.1	MiniCon-FD ⁺	113
6.3.2.2	MiniCon-FD ⁺⁺	115
6.4	Conclusion	123
7	Preference-based Composition of Web Services	126
7.1	Adopting MiniCon-FD ⁺ for Web Service Composition	126
7.1.1	Queries and Web services	127
7.1.2	Web Service Composition	129
7.2	User Preferences	134
7.2.1	Syntax of User Preferences	136

7.2.2	Semantics of User Preferences	138
7.2.3	Evaluation of User Preferences	142
7.3	Evaluation	145
7.3.1	Domain Ontology	145
7.3.2	Registered Web services	148
7.3.3	Queries and Preferences	150
7.3.4	Composition and Verification	152
7.3.5	Ranking the Composed Web Services based on user Preferences	156
8	Conclusions and Future Work	159
8.1	Research Contributions	160
8.1.1	Query Rewriting in the Presence of Full Dependencies	160
8.1.2	Preference-based Composition of DP Web Services	161
8.2	Future Work	162
	Bibliography	164

List of Tables

3.1	Registered Web services in the service registry	27
4.1	Description of dependencies	64
5.1	Ranked results of compositions	83
5.2	Related works evaluation	85
7.1	Registered Predicates in domain ontology	146
7.2	Domain ontology rules	147
7.3	Registered Web services in the service registry	149

List of Figures

2.1	SOA roles and operations [72]	11
2.2	Semantical languages for describing Web services [17]	15
2.3	An example of situation calculus [78]	17
2.4	Planning using graphplan [69]	19
3.1	High level view of our travel domain	25
5.1	Induced Preference Graph [79]	72
5.2	Graphical representation of the query	79
5.3	Syntax and semantics of FLE	80
6.1	Δ -graph basic structure	91
6.2	Converting a path to C-rule	99
6.3	A simple Δ -graph	102
6.4	Possible D-RADs in the Δ -graph	114
7.1	Relations between query subgoals and rewritten query subgoals	139
7.2	The Δ -graph of case study	148

Acknowledgements

Foremost, I would like to sincerely thank my supervisors, Dr. Adrian Fiech and Dr. Joerg Evermann, for their endless support and guidance. This research would not have been possible without their insightful advice and patient encouragement. I could not wish for a better or friendlier supervisor.

I would also like to acknowledge the financial, academic, and technical support of the Department of Computer Science and Memorial University. This research is financially supported by my supervisors' grant and the scholarship that I have received from the School of Graduate Studies of Memorial University.

Finally, I would like to thank my best friend, my wife, who gave me unconditional support and encouragement throughout this process. My love and thanks also go to my parents and my sisters for always being supportive of my academic pursuits regardless whether abroad or at home.

Chapter 1

Introduction

1.1 Motivation

Software systems are now an integral part of industry and play an essential role in every corner of companies' operations. As businesses grow, reducing financial cost and time of developing software as well as increasing compatibility to changes become more crucial from a software engineering prospective. Consequently, Service-Oriented Architecture (SOA) have been proposed to satisfy the current needs of businesses.

SOA is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains [29, 68, 42]. This architecture promises to leverage software systems to become more efficient and responsive to changes through service reuse and process agility [39]. In SOA, services are defined as software resources with externalized service descriptions. These descriptions are available for searching and binding a service consumer. Service consumers can either directly use the Uniform Resource Identifier (URI) for a service description, or can

use a service registry to find a service description. In addition to the role of service consumer, there are two other primary roles: the service provider, who implements services and publishes their corresponding descriptions, and the service registry that provides and maintains the repository of services.

Web services, as the practical examples of SOA, are modular, self-describing, self-contained applications that are accessible over the Internet by using standard Web protocols [72]. Web services can be classified into two groups based on their functionality: Data Providing (DP) Web services and state-changing Web services. DP Web services only provide information, while the state-changing services not only provide information, but may also change the state of the world. For example, a service that provides a list of departure flights from a given airport is classified as a DP service, and a flight booking service that can change the number of available seats (changing the state of the world) is a state-changing service.

SOA includes three main operations to support service consumers, providers, and registries: service discovery, service matchmaking, and service invocation. Service consumers search through the providers' catalogs for their desired Web service (service discovery). If they find a Web service that fulfills their needs (service matchmaking), they can bind it to their system (service invocation). However, no single Web service may be available to satisfy service consumers' needs. In such cases, there may be a chance to generate the desired Web service by composing available Web services. However, automatic composition of Web services in order to fulfill consumers' needs is a challenging and complex task, which is known to be an NP-hard problem [52, 70].

The results of the Web Service Composition process might be thousands of Web services, among which finding the desired Web services may be a difficult and time-

consuming task. To support service consumers, the composed Web services can be ranked based on the consumers' preferences, which are secondary objectives that should be satisfied as much as possible. Although different approaches have been proposed to address the Web Service Composition problem [28, 77], few of them consider the consumers' preferences in the composition system.

1.2 Approach

The goal of this thesis is to compose DP Web services and rank the composition results based on the preferences of service consumers, all in the presence of domain ontology. Domain ontology is an encoded source of knowledge about a specific domain. It provides a unified context to describe queries, services' descriptions, and consumers' preferences. To reach our goal, various methods for Web Service Composition are considered including AI-planning (finding a sequence of stored actions such that the preconditions hold and the goal is reached) and query rewriting (rewriting the received query in terms of some available data sources). Since our focus is on DP Web services, the query rewriting method is sufficient to effectively compose Web services.

Query rewriting originally belongs to the data integration field in which the goal is to combine physically separated sources of data to answer queries. In data integration, approaches are classified into Local-As-View (LAV) and Global-As-View (GAV). In LAV, each source of data is called a view. Views have their descriptions to specify their requirements as well as the information they provide. These descriptions and queries are described by the same language. Given a set of views, a LAV-based system rewrites the query in terms of views by composing some of the existing views

to answer the query. Various algorithms have been proposed to rewrite queries in LAV, including the MiniCon algorithm.

By having more knowledge about the domain on which the system relies, we can possibly have more rewritten queries. For instance, assume that the domain is about family relationships. Let one of the sources (views) provide a list of mothers who work in Memorial University, and let the given query ask about a list of parents who have a job. If the system knows that all mothers are also parents, then it can use this view to answer the query. To enable this capability, this type of knowledge should be first encoded and then utilized during the rewriting process. This problem is known as *query rewriting in the presence of dependencies*.

MiniCon is not capable of producing this class of rewritings. To have such a powerful system, the MiniCon algorithm needs to be extended, but we first need to encode this class of knowledge. Logical languages can be effectively used to fulfill this need. Using an expressive logical language allows more knowledge to be encoded; however, expressivity brings computational complexity. For example, even though tuple-generating-dependencies (TGD) is known as a useful tool to specify an ontology [15], their use in the query rewriting process results in undecidability [10].

To address this complexity issue in our approach, human knowledge is encoded using a sub-class of TGD, called full dependencies. We then extend the MiniCon algorithm to integrate views in the presence of full dependencies. We also adapt this extended algorithm to generate composed Web services. Finally, we design a formal framework to capture the service consumers' requests as well as their preferences to rank the composition results.

1.3 Organization of the Thesis

The rest of this document is organized as follows: In Chapter 2, the definition and characteristics of SOA as well as the role of people/software in this architecture are discussed. In addition, this Chapter reviews the definition of Web services, the importance of semantics in Web service composition, and the current methods for composing Web services. Chapter 3 describes a running example, which is used for evaluation of related works and our approach. Chapter 4 commences with describing data integration and its techniques and continues by classifying the algorithms for query rewriting. Finally, this Chapter concludes with comparison of these algorithms. Chapter 5 represents the related works and their evaluations, which are categorized in two groups of approaches related to *preference-based Web service composition* and *query rewriting in the presence of dependencies*. In Chapter 6, additional concepts are introduced as the basis for handling dependencies; these concepts are later used to extend the MiniCon algorithm. The proof of the correctness of this algorithm is provided as the conclusion of this Chapter. In Chapter 7, the adoption of the proposed algorithm for Web service composition problem is described. Moreover, our formal framework to handle user preferences as well as its evaluation are explained. Finally, our conclusion and our perspective for future works are described in Chapter 8.

Chapter 2

Service-Oriented Architecture and Web Services

2.1 Software Architecture

A software architecture involves the descriptions of the structures from which systems are built (software components), the relationships between the components, the principles and guidelines governing their design, and evolution over time. It also provides a description about how the information passes among the components [72].

In essence, an architecture is a plan which forms a backbone for building successful systems that meet well-defined requirements and possess the characteristics needed to meet those requirements. A software architecture [49] encompasses the significant decisions about:

- the organization of a software system,
- the selection of the structural elements and their interfaces by which the system is

composed, together with their behavior as specified in the collaboration among those elements,

- the composition of these elements into progressively larger sub-systems, and
- the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition.

Software architecture deals with the structure and the behavior of software as well as functionality, performance, re-usability, and technological constraints and tradeoffs. The fundamental purpose of software architecture is to efficiently manage the complexity of software systems in order to be appropriately modifiable in response to the changes in the business and technical environments. To reach this purpose, different software architectures such as SOA are introduced.

2.2 Service-Oriented Architecture

In the literature, different definitions of SOA given by different associations and companies can be found. For instance, OASIS [68] defines SOA as:

“A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.”

Another popular definition is given by IBM [29]:

“Service-oriented architecture (SOA) is a business-centric IT architectural approach that supports integrating your business as linked, repeatable business tasks, or services.”

As these definitions indicate, the core components of SOA-based software resources are packaged as “services”, which are well-defined, self-contained modules that provide specified business functionalities, and are independent of the state or context of other services [71]. The detailed explanation of characteristics of services is provided in Section 2.2.3.

2.2.1 Principles of SOA

In order to enable SOA to gain its promises, a set of principles should be obeyed. According to the literature [29, 72, 17], a mutual set of principles can be specified. These principles are as follows:

- **Scalability:** SOA should work in a variety of settings such as within an organization, between business partners, and across the world.
- **Loose coupling:** SOA is an evolution from tightly coupled systems to loosely coupled ones. Service requester and service provider should be independent of each other.
- **Interoperability:** One service/application should be able to communicate with others regardless of the machines they are running on.
- **Discovery and invoking:** Services can be dynamically discovered, invoked, and (re-) combined. This is accomplished through directory of service descriptions.

- **Abstraction:** This principle emphasizes the need to hide as much of the underlying details of a service as possible. Instead of focusing on systems and applications, developers can concentrate on building services for business users. Doing so directly enables and preserves the previously described loosely coupled relationship.
- **Standards:** Interaction protocols must be standardized to ensure the widest interoperability among unrelated service providers. Contracts should also be standardized. Furthermore, standards are the basis of interoperable contract selection and execution.
- **Self-Healing:** The ability to recover from errors without human intervention during execution.

2.2.2 Roles of Interaction in SOA

There are three primary groups that play significant roles in SOA [29, 72]: service provider, service registry, and service consumer. A service provider is an organization (from business point of view) or a platform (from architectural point of view) that owns the service and implements the business logic that underlies the service, host, and controls access to the service. Service providers are responsible for publishing their services in a service registry hosted by a service discovery agency. This involves describing the business information, technical information of the service, and registering that information in the service registry in the format prescribed by the discovery agency.

A service consumer is the enterprise (from business point of view) or the application

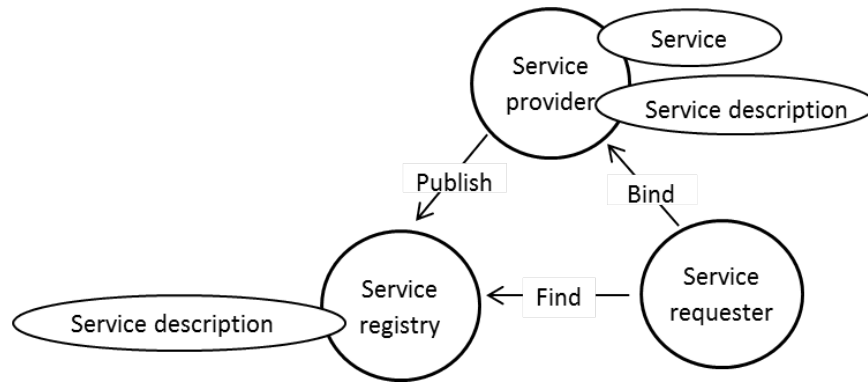


Figure 2.1: SOA roles and operations [72]

(from architectural point of view) which looks for those services that provide certain functions and satisfy the consumers' needs. The service consumer searches for the desired services among the service descriptions that are already provided by different service providers and are registered in the service registry. After finding the desired service description, the service consumer uses the information in the description to bind the service. A service consumer can be either a browser driven by an end user, or another service as part of an application.

To assist the mentioned groups in their roles, service registries can be defined. A service registry is a searchable directory, where service descriptions can be published and searched. A service discovery agency is responsible for providing the infrastructure required to enable the three operations described above in SOA: publishing the services by service providers, searching for services by service consumers, and invoking (binding) the services by service consumers.

2.2.3 Services as the Core Component of SOA

In SOA, the entire functionalities of an enterprise are decomposed into smaller, distinct units of functionalities. To retain independency between services, services encapsulate some of the functionalities within a distinct context. Hence, the size of a service (granularity) can vary. A service can be:

- a self-contained business task, such as a funds withdrawal or funds deposit service,
- a full-fledged business process, such as an automated purchase of office supplies,
- an application, such as a life insurance application or demand forecasts and stock replacement, or
- a service-enabled resource, such as an access to a particular back-end database containing patient medical records.

Furthermore, a service can encompass the functionalities provided by other services. In this case, one or more services are composed together. In order to compose services, or in the case that a service requests another service, they need to have a relationship. According to the principles of SOA represented in Section 2.2.1, these relationships should be loosely coupled. Therefore, these relationships are achieved through the use of service descriptions by messaging.

Each service in SOA should possess the following characteristics:

- **Loose coupling:** Services maintain a relationship that minimizes dependencies and only requires that they retain an awareness of each other.
- **Service contract:** Services adhere to a communication agreement, as defined collectively by service descriptions.

- **Autonomy:** Services have control over the logic they encapsulate, and their logic cannot be changed from outside world.
- **Abstraction:** Beyond what is described in the service contract, services hide logic from the outside world.
- **Reusability:** Logic is divided into services with the intention of promoting reuse.
- **Composability:** Collections of services can be coordinated and assembled to form composite services.
- **Discoverability:** Services are designed to be outwardly descriptive so that they can be found and accessed via available discovery mechanisms.

Services form a special subset of services in SOA architecture, and they are the focus of this study. As such, the next Section is devoted to discuss Web services as well as the need for semantics when dealing with them.

2.3 Web Services

Web services are modular, self-describing, self-contained applications that are accessible over a network like the Internet [21]. They are the most popular realization of the SOA, and they should hold all of the characteristics of services represented in Section 2.2.3.

A Web service description has two major interrelated components which are functional and non-functional characteristics. The functional description details the operational characteristics, i.e., details about the overall behavior, how the service is invoked,

and the location where it is invoked. The non-functional description concentrates on service quality attributes such as service cost, performance, security, availability, and response time.

Web service descriptions are described in terms of a description language. As mentioned in Section 2.2.2, one of the operations in SOA is service searching that is divided into two sub-operations: service discovery and service matchmaking. Service descriptions are used to perform these two operations. Finding a description language that satisfies the operations' requirements is a challenging task. A language that only consists of a set of terms is not an effective solution due to the inherent ambiguities in natural languages [32]. For instance, using the term `bookTrip` to describe the functionalities of a Web service brings some ambiguities such as the type of the trip (e.g. it is a flight trip or a train trip) that is unspecified. Even after clarifying the meaning, how to use this Web service, such as the required input or the type of output, is still unclear. Hence, in order to provide an effective description language, a semantic solution is required.

2.3.1 Semantics of Web Services

One of the useful tools to incorporate semantics to the Web service description is ontology [31, 64]. Ontology is defined as a formal and explicit specification of a shared conceptualization [34]. Ontologies aim to construct a shared and common understanding of terms and relations between them for a given domain across people, organizations, and application systems [86] (For more information about ontologies in semantics, readers are referred to [59]).

Proposal	Comment	Ontology Language	Formalism
OWL-S Process Model	An OWL based upper ontology for representing Web processes.	OWL	Description Logics
FLOWs	A combination of First Order Logic and rules to represent Web services. Based on Process Specification Language (PSL). FLOWs rectifies the problems with OWL-S process model.	SWSO (FLOWs + ROWs)	First-Order Logic
WSMO Orchestration	The orchestration model in WSMO refers to complex services with states.	WSML	Description Logics, First-Order Logic and Logic Programming (F-Logic)

Figure 2.2: Semantical languages for describing Web services [17]

Ontologies are expected to play a central role to empower Web services with expressive and computer interpretable semantics. The combination of Web services and ontologies has resulted in the emergence of a new generation of Web services called semantic Web services [13]. To provide semantics in Web services using ontologies, different languages have been used such as OWL-S, FLOWs, and WSMO. Integrating ontology into Web services may not only enhance the quality and robustness of Web service management, but also pave the way for semantic interoperability. Semantic Web services play an essential role in industry and academia to address challenging research issues such as automatic selection, monitoring, and composition of Web services.

2.3.2 Web Service Composition

Web Services are composed when no single Web service can satisfy functionalities requested by a service consumer since there might be a possibility to combine existing services to fulfill the request. As it is beyond the human capability to deal with the composition manually, various approaches from different perspective have been

proposed to automate this process.

Despite these efforts, WSC is still a highly complex task, and is known to be an NP-hard problem [52, 77]. The increasing number of available services over the Internet as well as the need to handle potentially frequent updates of services at runtime has led to such a complexity. In addition, the lack of a unified language to describe and evaluate Web services, which can be developed by different organizations, has made this problem even more challenging. Consequently, automatically composing the Web services is vital. To facilitate an automatic composition process, semantics plays a significant role. Two main methods which have been adopted in the semantic-based approaches are AI-planning and query rewriting.

2.3.2.1 Web Service Composition via AI-planning

Planning is the problem of synthesizing a course of action that, when executed, will take an agent from a given initial state to a desired goal state [43]. In general, a planning problem consists of a description of an initial state, a goal state, and a set of actions [73]. For a given problem instance, the planning task is to generate a sequence of actions that when performed from the initial state, will terminate in the goal state [62].

Although it is shown that using AI-planning for Web service composition cannot reduce the computational complexity of this problem [70], many efforts have been made to adopt it for this problem because of the similarities that exist between them. The [73] is a survey on classifying AI-planning methods for Web service composition. The categories which are discussed in the Related Works (Chapter 5) are described in the rest of this Section.

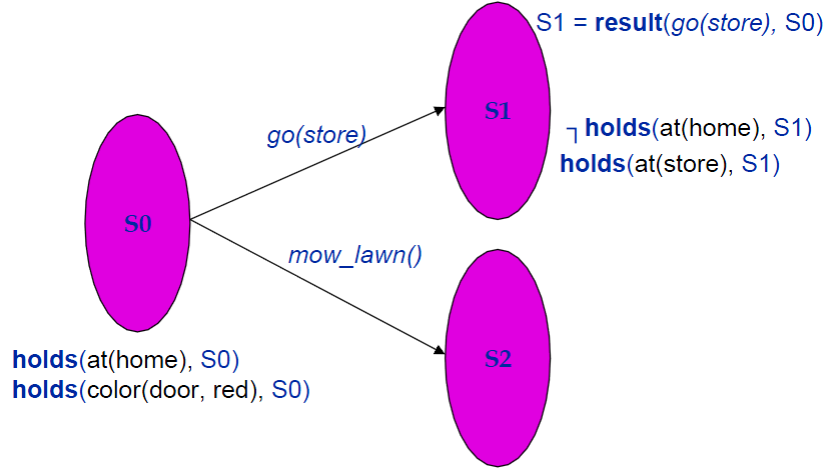


Figure 2.3: An example of situation calculus [78]

Situation calculus Situation calculus is a logical language for specifying and reasoning about dynamical systems [78]. The key idea in situation calculus is to represent a snapshot of the world, called a ‘situation’ explicitly. A situation is expressed in terms of functions and relations called fluents. Fluents are statements which are true or false in any given situation, e.g. ‘I am at home’. Another component in situation calculus are actions which map situations to situations. For instance, as Figure 2.3 shows, ‘go’ and `mow_lawn()` are actions; ‘holds’ and ‘result’ are fluent in this example.

Golog is a logic programming language built on top of the situation calculus. Some approaches try to adapt and extend the Golog language for automatic composition of Web services [64, 63]. The general idea underlying this method is that software agents can reason about Web services to perform automatic Web service discovery, execution, composition and inter-operation. The user’s requests and constraints can be presented by the first-order language and the situation calculus (a logical language for reasoning about action and change). The authors conceive each Web service as

an action - either a primitive action (i.e., changing the state) or a complex action (i.e., compositions of primitive actions). The agent knowledge base provides a logical encoding of the preconditions and effects of the actions (Web services) in the language of the situation calculus. The agents use procedural programming language constructs which are composed with concepts defined for the services and constraints defined using deductive machinery. A composite service is a set of atomic services which are connected by procedural programming language constructs (if-then-else, while, and so forth). Since Golog is a programming language on the top of situation calculus, it provides flexibility to encode complex goals. Moreover, non-deterministic domains can be addressed by this method; however, using this method to find the shortest plan is not guaranteed [73].

Graph-based planning Several planners discussed so far utilize graph structures for the extraction of heuristics. A planning graph is a directed leveled graph which consists of two types of nodes, namely action nodes and proposition nodes. These nodes are arranged in alternating levels consisting of proposition nodes followed by layers of action nodes and so forth (e.g., in Figure 2.4 nodes in levels zero, two, and four are proposition nodes, while others are action nodes). A planning graph consists of a sequence of levels that correspond to time steps. Figure 2.4 shows a simple example of a planning graph. The initial level (zero level) of a planning graph is a proposition level which consists of one node for each proposition of the initial situation (in this example, A, B, and C are considered as proposition). The second level (level 1) is an action level which contains all actions whose preconditions are satisfied by the proposition nodes of the initial level. The third level (level 2) is again a proposition level, containing the proposition nodes from level 1 and proposition nodes

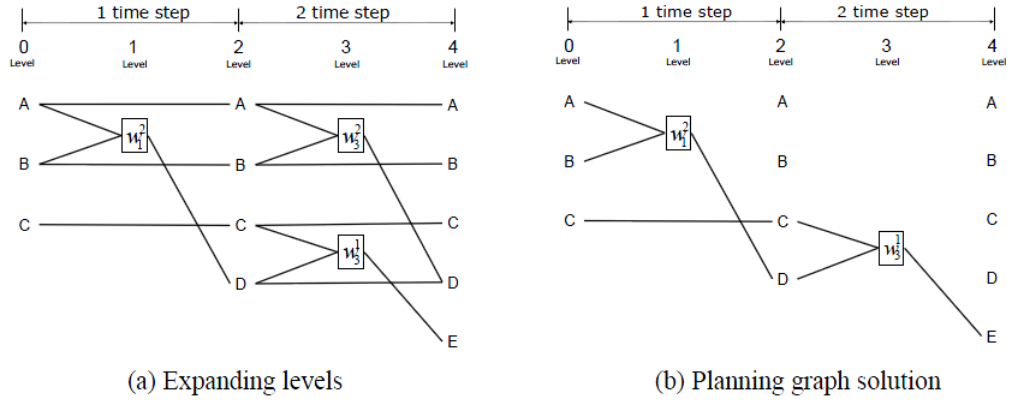


Figure 2.4: Planning using graphplan [69]

that represent the effects of the actions of the preceding action layer. All actions at some level i are connected to the preconditions at level $i-1$ and its effects at level $i+1$, introducing or negating proposition in $i+1$. The construction of the planning graph stops when two consecutive propositional layers are identical. It has been shown that this step always occurs, guaranteeing the termination of the process. The complexity of creating a planning graph is low-order polynomial in the number of actions and propositions [73]. One of the well-known planners using this method is Graphplan [12]. The advantage of Graphplan is its better performance compared to Golog. Moreover, the soundness, completeness, generation of shortest plans, and termination on unsolvable problems are proven. However, Graphplan cannot handle conditional or universally quantified effects. In addition, Graphplan will have performance issues in domains in which too much irrelevant information is encoded in the specifications of a planning task [73].

Planning as Satisfiability The idea behind the planning as satisfiability-approach is to express the planning problem as a reasoning problem for which powerful problem solving algorithms exist. Effects of an action are implied by the occurrence of the ac-

tion when its preconditions hold. Planning is then formalized as the process of finding a deductive proof of a statement that asserts that the initial conditions together with a sequence of actions imply the goal condition [30]. Different types of logic such as propositional logic and description logics can be used as the core of this method [73]. SATPLAN [45] is the implemented algorithm based on this method. Blackbox [44] is the mixture of SATPLAN and Graphplan algorithms. The benefits of this method is that more logical operations can be defined to describe effects/situations.

HTN Planning Hierarchical Task Network (HTN) planning [61] is a well-known planning paradigm that has been employed for Web service composition problem (e.g., [56, 82, 81, 80]). In this approach, a method is defined as a description of how tasks can be decomposed. Given an initial state, an initial task network (the objective of planning), and a domain description comprising of a set of operators and methods, an HTN planner constructs a plan by repeatedly decomposing tasks into smaller and smaller sub-tasks until a primitive decomposition of the initial task network is found. In more formal definition, an HTN planning problem is a 3-tuple $P = (s_0, w_0, D)$ where s_0 is the initial state, w_0 is the initial task network, and D is the HTN planning domain which consists of a set of operators and methods. A task network is a pair $w = (U, C)$ where U is a set of task nodes and C is a set of constraints. The constraints normally considered are of type precedence constraint, before-constraint, after-constraint or between-constraint. $\pi = o_1 o_2 \dots o_k$ is a plan for HTN planning program $P = (s_0, w_0, D)$ if there is a primitive decomposition, w , of w_0 of which π is an instance. This method is powerful to deal with the large and complex domains; however, it may not deal with dynamic environment due to its need to receive tasks with explicit descriptions.

2.3.2.2 Web Service Composition via Query Rewriting

Query rewriting method originally belongs to the data integration field, in which the purpose is to answer a given query by integrating the available sources of data. Thus, a description should be assigned to each source, using which the integration system can create a plan for combining sources. To enable the system to compare the descriptions and to produce an integration plan that satisfies the submitted query, the sources' descriptions and consumers' queries are required to be described over the same language, called global schema. Different logical languages can be used for the global schema such as datalog rules and various description logics [37].

Data integration systems are defined as a triple $\langle G, S, M \rangle$, where G is the global schema, S is the set of data sources, and M is a set of mappings between sources and the global schema. Queries are defined by using the set G . The techniques for rewriting a query differ significantly and depend on the database integration approach that is used, e.g., Global-As-View (GAV) or Local-As-View (LAV) [53].

The difference between these two techniques results from the difference in their mapping function. In GAV, each member of G is mapped to conjunctions of elements in S . This type of mapping makes the query rewriting process simple. To rewrite queries in GAV, it is enough to replace the names in the query by their mappings in S . However, these mappings make GAV unsuitable for the domains where sources' definitions and their providing data can change. When a change in a source occurs, all the mappings in which the source appears need to be revisited/updated. Therefore, these re-investigations reduce the performance of the system.

LAV has a different strategy for mapping: each member of S is mapped to conjunction

of some elements of G . This strategy enables the system to be more compatible in response to changes in data sources. When a change in a source occurs, only the mapping of S needs to be updated which is actually updating the conjunction. Nevertheless, the query rewriting in this technique is highly complex [36]. Since LAV is more suitable for source-changing domains in contrast to GAV, it can be adapted for Web service composition problem. Therefore, various LAV-based approaches have been proposed for Web service composition [58, 8, 65, 7, 87, 11].

In this work, our focus is on data-providing Web services which can get some data as input (possibly empty input) and essentially provide some data as output. Therefore, they have no state changing after calling. Although data-providing services can be described using AI-planning techniques, all the capabilities of these techniques such as encoding the effects after calling Web services or state changing may not be utilized. Therefore, some investigations in these techniques may not be used during the composition process. On the other hand, data-providing services can be seen as data sources such that their definitions may change iteratively by their providers. Hence, our approach is focused on adopting LAV to Web service composition.

Chapter 3

Running Example

Several approaches have been proposed to compose Web services in the presence of user (i.e., service consumer) preferences. Different techniques such as AI-planning and query rewiring are adapted to compose Web services. Besides, various logical languages including propositional logic, temporal logic, and description logics are used to capture user preferences. Due to these variations in proposed methods, comparing different approaches is a difficult task. In order to conduct such a comparison, an example is first described. Here, this example is explained in a high-level description, which is independent of the underlying technologies and/or languages. Later, these descriptions will be revisited and will be expressed with respect to the languages discussed in related works as well as the language used in our approach. This example will provide us with a unified context to discuss related works, shown in Section 5.1.1), and to evaluate our approach, which is explained in Section 7.3.

Our running example is based on the travel domain which has been the domain of focus in many Web service composition studies. In this example (i) the knowledge

regarding the travel domain is structured, (ii) a set of Web services are assumed, and (iii) some scenarios are described in which a Web Service composition system needs to be run.

3.1 Structure of the Domain

A widely used technique to describe a domain is through using an ontological structure. The most well-known definition of ontology is given by [34], where an ontology is described as "a specification of a conceptualization". In general, ontology consists of three main components: a set of terms, the meaning of terms, and the semantical relations between the terms. Since an ontology consists of terms with fixed meanings, it is required to declare terms used in our example's domain.

Our domain of focus consists of three sub-categories which are (i) *Location*, (ii) *Trip* and (iii) *Accommodation*. The *Locations* category is classified based on the actual geographical locations. Term *World* refers to any geographical location in the world. Term *City* can refer to any city. A more detailed example is an *African City* that refers to the cities located in Africa. Similarity, the term *European City* refers to the European cities; and *Swedish City* refers to Swedish cities. A *Trip* is a pair of locations *A* and *B* such that a movement from *A* to *B* is possible. This movement can be done by plane or train; thus, *Trip* consists of two sub-categories *Flight* and *Train*. For example, moving from St. John's to Toronto by plane is an instance of *Flight*, and consequently it is a trip. A trip is called round-trip if there is a movement from *A* to *B* and another movement from *B* to *A*. The *Accommodation* category includes Hotel, B&B, and Motel. Hotels are classified based on their prices to 3-star, 4-star,

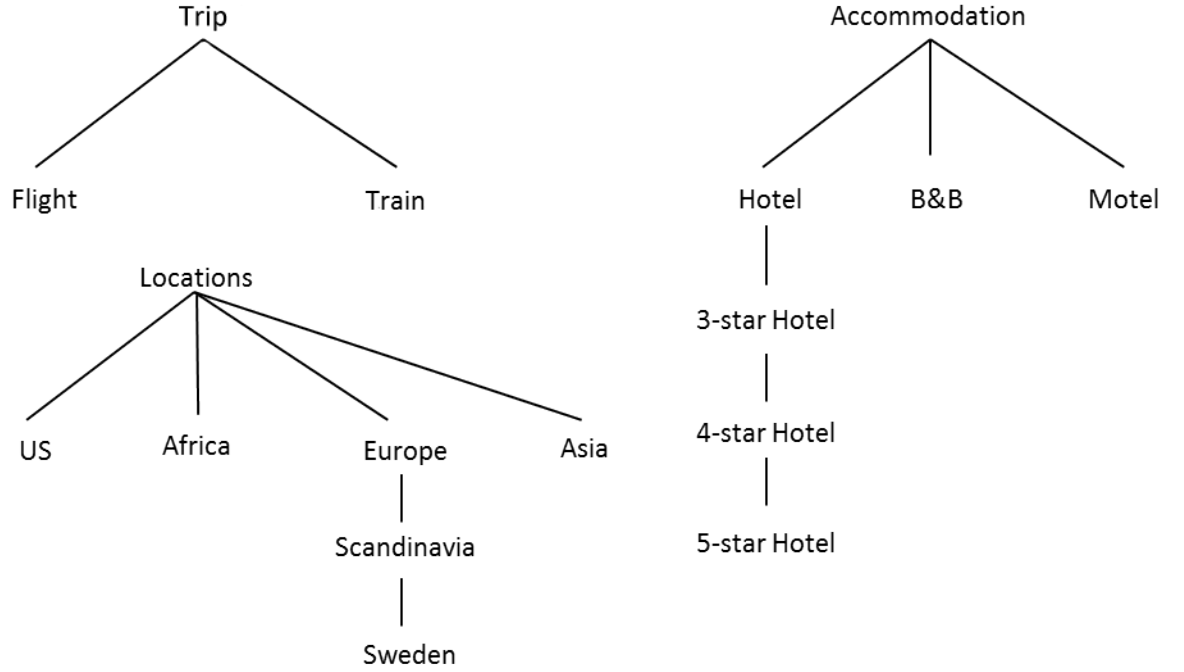


Figure 3.1: High level view of our travel domain

5-star hotels. We assume that if customers can afford a 5-star hotel, they can afford a 4-star hotel as well. We assume the same for 4-star and 3-star hotels. The high-level view of this domain is illustrated in Figure 3.1.

3.2 Available Web Services

We assume that there exists a set of Web services that are registered in a service registry. In this thesis, a Web service is considered as a component that provides some data as its output when its required data is provided as its input. As mentioned in Section 2.3, each Web service requires a description of its functionalities; thus, in our example, the following structure is specified for a Web service description:

- service input, which declares the required data format for the service to run,
- service output, which specifies the format of the data provided by the service,
- and
- a detailed information about the output data and input data and their semantical relations.

The language of these descriptions should be borrowed from the language of the service registry; thus, for now, we leave them as English statements. In Section 7.1.1, we will formally translate them to a proper logical language

The service registry is responsible for storing the descriptions, which assigns a unique name to each registered Web service and then stores the information regarding the location of the Web service and about how to invoke it. These responsibilities are out of the scope of this thesis. We assume we have a unique name for the Web services by which the physical Web services can be invoked. Our focus is to find a logical language to properly describe the functionalities of Web services and later use these logical descriptions to perform Web service composition. For ease of readability, terms in the ontology are used to describe the inputs and outputs of the Web services. The registered Web services in our assumed service registry are listed in Table 7.3.

NO	Name	Input	Output	Description
1	ETrain	European city c	Train trips	provides a list of train trips (c,x) where x is located in Europe
2	Scandinavia- Train	Scandinavian city c	Train trips	provides a list of train trips (c,x) where x is located in Scandinavia
3	EFlight	European city c	Flight trips	provides a list of flight trips (c,x) where x is located in Europe
4	LocalSweden Airlines	Swedish city c	Flight trips	provides a list of flight trips (c,x) where x is located in Sweden
5	WestJet	Any city c	Flight trips	provides a list of flight trips (c,x) where c and x can be any location
6	EgyptAirlines	African city c	Flight trips	provides a list of flight trips trips (c,x) where x is located in Africa
7	AirCanada	Any city c	Flight trips	provides a list of flight trips trips (c,x) where c and x can be any lo- cation
8	StarHotel	Any city c	4-star hotels	Provides a list of 4-star hotels in c
9	AAHotel	Any city c	3-star hotels	Provides a list of 3-star hotels in c
10	RoyalHotel	Any city c	5-star hotels	Provides a list of 5-star hotels in c

Table 3.1: Registered Web services in the service registry

3.3 Scenarios

Let us assume a travel reservation agency is planning to provide various offers to its customers. The functionalities of the Web services and their preferences are as follows:

- 1) Desired Web service 1 (Simple trip): the Web service gets a city c_1 as input and provides a list of cities c_2 such that moving from c_1 to c_2 is possible.

Preference: Trips preferred to be flight.

- 2) Desired Web service 2 (Fast one-stop round-trip travel): the Web service gets a city c_1 in the world and provides three cities c' , c'' , and c_2 such that moving from c_1 to c' , c' to c_2 , c_2 to c'' , and c'' to c_1 is possible (round-trips with one stop per leg).

Preference: Clients prefer to travel by plane. This preference is more important for the return legs because clients usually desire to reach their home as fast as possible. Thus, trips preferred to be flight.

- 3) Desired Web service 3 (Visiting 4 European cities): it gets a city c_1 and provides the names of four European cities c_2 , c_3 , c_4 , and c_5 such that there exists a trip from c_1 to c_2 , c_2 to c_3 , c_3 to c_4 , and c_4 to c_5 .

Preference: If two consecutive cities are in Scandinavia, the travel between them should be done by train because of the lower price and the beautiful sceneries.

- 4) Desired Web service 4 (Travel around the world): it gets a city c_1 in the world

as the input and provides four city names c_2 to c_5 such that there exist a trip from c_1 to c_2, \dots, c_4 to c_5 , and c_5 to c_1 .

Preference: At least three of the stops are preferred to be in Africa.

- 5) Desired Web service 5 (Affordable travel): This Web service gets a city c and returns a list of (trip, hotel) such that the trip originates from c , and the hotel is located in the destination.

Preference: If the destination is located in Europe, the room is preferred to be in a 3-star hotel.

In Chapter 5, the related works along with the results of the evaluation performed based on our running example are provided. The results of evaluation imply that current approaches lack the ability to describe some of the above preferences and some shortcomings exist in their ranking system. We later, in Chapter 7, propose a formal system to address these shortcomings.

Chapter 4

Data Integration and Query

Rewriting

Data integration is the problem of answering queries by combining different data sources and providing unified views of these sources that are physically distributed over different locations. For example, consider a query about a movie, actors of the movie, the theatres where the movie is being played, and the reviews about the movie. Let IMDB, Empire theatre database, and iTunes movie trailers website be the sources that we have. Now, the question is how to integrate these data sources to answer the query.

Since data sources can be produced anonymously, a mediated schema is required to virtually create unified views for the sources. The required schema, which is called global schema, enables an integration system to compare sources and reason on queries (e.g., checking whether two queries are equivalent). To provide unified views of sources, a set of relations between the global schema and sources is required. The

mechanism of describing these relations distinguishes the techniques of integrating data.

Two well-known techniques have been proposed for defining these relations: Global-As-View (GAV) and Local-As-View (LAV). GAV expresses the global schema in terms of the data sources, while the second approach, LAV, specifies the global schema independently from the sources, and then expresses the sources in terms of the global schema. These expressions are called views of sources. The formal definition of an integration system is provided in definition 4.0.1.

Definition 4.0.1 Data integration system. *A data integration system is defined as a triple $\langle G, S, M \rangle$, where G is the global schema expressed in a language \mathcal{L} , S is the set of data sources, and M is a set of mappings between sources and the global schema. \square*

Various languages have been used for describing queries and views such as conjunctive queries without and with inequalities, positive queries, datalog, and first-order queries [36]. Most of the works in data integration have been done using datalog and conjunctive queries which are logic-based database languages [37]. Hence, the syntax and semantics of these two languages are first reviewed, and then the techniques of integrating data are explained in detail.

4.1 Datalog

A datalog [37] rule is a logical implication that may only contain conjunctions, constant symbols, and universally quantified variables, but no disjunctions, negations,

existential quantifiers, or function symbols. Datalog is considered a sub-language of first-order logic to which the classical semantics applies. The syntax and the semantics of datalog rules are described in the definitions 4.1.1 to 4.1.4.

Definition 4.1.1 (Datalog Syntax) *The syntax of datalog consists of the sets $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$ such that \mathbb{C} is the set of constant symbols, \mathbb{V} is the set of variable symbols, and \mathbb{P} is the set of predicate symbols. Each predicate has a fixed number as its arity. The sets \mathbb{C} and \mathbb{P} are usually assumed to be finite, while the set \mathbb{V} can be countably infinite. The union of the sets \mathbb{C} and \mathbb{V} is also called datalog terms. \square*

Definition 4.1.2 (Datalog atom) *Given a datalog syntax $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, the formula $P(t_1, \dots, t_n)$ is a datalog atom such that $P \in \mathbb{P}$ has the arity n , and t_1, \dots, t_n are datalog terms. In addition to these atoms, any datalog syntax also has two default atoms, shown by \top and \perp , with arity zero. \square*

Definition 4.1.3 (Datalog rule) *Given a datalog syntax $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, a datalog rule is an expression of the form*

$$H(u) : -P_1(u_1), \dots, P_n(u_n) \quad (4.1)$$

where $H(u)$ and $P_i(u_i)$ are datalog atoms; $H(u)$ is called the head; $P_1(u_1), \dots, P_n(u_n)$ is called the body; and the comma shows the conjunction between the atoms $P_i(u_i)$. \square

Definition 4.1.4 (Datalog semantics) *Given a datalog syntax $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, the datalog semantics is defined by a datalog interpretation $I(.^I, D^I)$ where D^I is an interpretation domain and $.^I$ is an interpretation function; the domain is an arbitrary set that defines the world objects, and the interpretation function establishes the mapping from symbols into this domain such that*

- If $c \in C$ is a constant, then $c^I \in D^I$
- If $P \in \mathbb{P}$ is a predicate symbol of arity n , then $P^I \subseteq (D^I)^n$.

Now, for a given variable assignment function $Z : \mathbb{V} \rightarrow D^I$, the interpretation function $.^I$ can be extended as follows:

- If t is a term, then its interpretation is defined as:

$$t^{I,Z} = \begin{cases} t^I \in D^I, & \text{if } t \in \mathbb{C} \\ Z(t) \in D^I, & \text{if } t \in \mathbb{V} \end{cases} \quad (4.2)$$

- If $P \in \mathbb{P}$ is a predicate symbol of arity n , then $P^{I,Z} \subseteq (D^I)^n$.
- If \top and \perp are datalog atoms with arity zero, $\top^{I,Z} = \text{true}$ and $\perp^{I,Z} = \text{false}$ for any variable assignment Z in I .
- For a predicate $P \in \mathbb{P}$ with arity n , $P(t_1, \dots, t_n)^{I,Z} = \text{true}$ if $(t_1^{I,Z}, \dots, t_n^{I,Z}) \in P^I$; otherwise, $P(t_1, \dots, t_n)^{I,Z} = \text{false}$.
- For a conjunction of datalog atoms B_1, \dots, B_n , $(B_1, \dots, B_n)^{I,Z} = \text{true}$ if $B_i^{I,Z} = \text{true}$ for all $i=1, \dots, n$; otherwise, $(B_1, \dots, B_n)^{I,Z} = \text{false}$.
- For a datalog rule $H :- B$ such that B can be an arbitrary conjunction of datalog atoms, $(H : -B)^I = \text{true}$ if for all possible variable assignments Z for I , either $B^{I,Z} = \text{false}$ or $H^{I,Z} = \text{true}$; otherwise, $(H : -B)^I = \text{false}$.

□

4.2 Conjunctive Queries

A conjunctive query [1, 74] is a datalog rule but with a unique head (see definition 4.2.1) predicate which is not in datalog syntax and with restrictions over the range

of its variables. The intuition behind these restrictions is to guarantee a finite result when a rule is applied to finite relations because checking whether a rule has a finite result is undecidable. This restriction is called *safe*, i.e., each variable occurring in the head must also occur in the body. As mentioned in definition 4.1.4, the semantics of datalog rules is model-theoretic. Hence, if a rule has a model with infinite size, then it is not tractable. For instance, in the unsafe rule $CarPrice(c, price) : \neg NewCar(c)$, the variable $price$ can be anything. Our understanding from a rule is that “if the body is true, then the head is true”. Therefore, for any data x , if x is a new car, the data $(x, price)$ should be inferred for any (possibly infinite) value for $price$. Consequently, building such a model is not tractable. For more explanations about safety and creating a minimal model readers are referred to the Section 12.3 of [1]. The formal description of syntax and semantics of conjunctive queries are represented in definitions 4.2.1 and 4.2.2.

Definition 4.2.1 (Conjunctive Query Syntax) *Given a datalog syntax $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, a conjunctive query is an expression of the form*

$$Q(t_1, \dots, t_k) : \neg P_1(\bar{u}_1), \dots, P_n(\bar{u}_n) \quad (4.3)$$

*where $Q \notin \mathbb{P}$ and $k \geq 1$; tuple (t_1, \dots, t_k) consists of \mathcal{L} – terms; each $P_i(\bar{u}_i)$ is an \mathcal{L} -atom; and $Vars(t_1, \dots, t_k) \subseteq Vars(\bar{u}_1) \cup \dots \cup Vars(\bar{u}_n)$ where $Var(\bar{x})$ denotes the set of variables occurring in \bar{x} . $Q(t_1, \dots, t_k)$ is called **head**, and the rest of the query is called **body**. Each atom in the body is also called **subgoal**. Variables which appear in the head are called *distinguished*, while the remains are called *non-distinguished* (also called *existential*) variables. \square*

Definition 4.2.2 (Conjunctive Query Semantics) *Given a syntax \mathcal{L} and an interpretation I , the semantics of a conjunctive query $Q(t_1, \dots, t_k) : -P_1(\bar{u}_1), \dots, P_n(\bar{u}_n)$ is defined as follows: if there is a variable assignment Z by which $P_i(\bar{u}_i)^{I,Z} = \text{true}$ for $i=1, \dots, n$, then $(t_1^{I,Z}, \dots, t_k^{I,Z}) \in Q^{I_Q}$. I_Q is then the extension of I to have an interpretation for Q based on the already specified interpretations of symbols in \mathcal{L} . Intuitively, $Q^{I_Q} \subseteq (D^I)^k$ contains k -tuples of objects resulting from all the possible variable assignment functions in which all the query subgoals are evaluated to be true. Therefore, for a given interpretation I and variable assignment Z , we will say $Q(x_1, \dots, x_k)^{I,Z} = \text{true}$ if $P_i(u_i)^{I,Z} = \text{true}$ for $i = 1, \dots, n$. \square*

Example 4.2.1 *Consider the datalog language $L = \langle \{\text{john}, \text{sara}\}, \{x, y, z, u, w, m, n, t, e, v_1, \dots, v_n\}, \{\text{Student}, \text{SupervisorOf}, \text{Happy}, \text{Smart}\} \rangle$ such that atoms $\text{Student}(x)$, $\text{Happy}(x)$, or $\text{Smart}(x)$ indicate whether x is a student, a happy person, or a smart one, respectively, and the atom $\text{SupervisorOf}(x, y)$ shows that the supervisor of x is y .*

Any interpretation I can be seen as a database where the set of all the objects in the database forms the D^I , and the interpretation of each constant is an object in D^I (e.g., real persons John and Sara who are registered in the database I). Each L -predicate's interpretation is a table such that the number and the order of columns are the same as the arity and the order of predicate's arguments.

For instance, we can have a table in the database called student with one column, where all the rows in this table are students, or a table SupervisorOf with two columns, where the second column's object is the supervisor of the object in the first column for each row in this table.

Also, someone may be interested to find all the students that are supervised by a/some happy person(s); thus, a conjunctive query $Q(x) :- Student(x), SupervisorOf(x, y), Happy(y)$ can be used to fulfill the request. This query can be answered by joining the above three tables; Q^{IQ} , the answer of the query, will be a new table with one column that contains all the students of happy supervisors. \square

Example 4.2.2 Let L be the language described in example 4.2.1, and let D^I be $\{Sara, John, Arash, Don, Edward\}$ and $SupervisorOf^I = \{(Arash, John), (John, Edward), (Sara, Don), (Don, Edward)\}$. Assume a query $Q(x, y) :- SupervisorOf(x, z), SupervisorOf(z, y)$ is received. To find possible answers, variables should be substituted with any possible object in D^I . This substitution is done by defining a variable assignment function Z . If such a variable assignment function exists by which all the subgoals of the query are evaluated to be true, we then apply this variable assignment function to the head and store it as one of the answer for the query.

Back to the example, the only variable assignment functions which make all the subgoals true are $Z_1 = \{x \rightarrow Arash, z \rightarrow John, y \rightarrow Edward\}$ and $Z_2 = \{x \rightarrow Sara, z \rightarrow Don, y \rightarrow Edward\}$, and they yield heads $Q(Arash, Edward)$ and $Q(Sara, Edward)$ which are the answers. \square

Up to this point, datalog and conjunctive queries and their syntax and semantics are described. These two languages are well-studied languages which are widely used in data integration approaches, and they are also used in our approach. Hence, in the next Section, the data integration techniques are explained by employing these two languages.

4.3 Data Integration Techniques

As mentioned in definition 4.0.1, a data integration system consists of a global schema, a set of data sources, and a set of mappings between the global schema and data sources. The differences between data integration techniques stem from the difference in defining the mappings. Two popular well-studied techniques have been proposed for data integration: Global-As-View (GAV) maps each term in the global schema to some data sources, and Local-As-View (LAV) which maps each source to some terms in the global schema [1].

In any data integration system, queries are described over the global schema, and the purpose of this system is to answer the queries by integrating some of the sources. This means exploiting the mappings to reformulate the query Q , which is in terms of global schema, into another query Q' which is in terms of sources and can be evaluated by considering the evaluation of data sources. This process is called query rewriting, and Q' is considered as a rewriting of Q .

In the rest of this Section, GAV and LAV as well as the algorithms for query rewriting based on these techniques are described, and their advantages and disadvantages are discussed.

4.3.1 Global-As-View (GAV)

Assume a data integration system $\langle G, S, M \rangle$ where G itself is $G = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$ (i.e., a datalog syntax), and S is a set of sources' names. In GAV, the set M is populated by mappings from global schema to data sources.

A mapping in a GAV-based system is a datalog rule that is defined with respect to

$\mathcal{L}' = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \cup S \rangle$ such that the predicate symbol used in the head of the rule belongs to the set \mathbb{P} while the body's predicate symbols are data sources' names.

For example, assume the predicate symbol *Flight* exists in \mathbb{P} which verifies flight trips between two cities. Let *AirCanada* and *WestJet* be the data sources that provide a list of flight trips between two cities, and their names are stored in S . The following possible mappings can be defined by using *AirCanada* or *WestJet* in form of datalog rules:

- $Flight(a, b) : \neg AirCanada(a, b)$
- $Flight(a, b) : \neg WestJet(a, b)$

As mentioned, a query in GAV is expressed in the terms of the global schema, i.e., \mathcal{L} . Queries can be described in the form of conjunctive queries defined in Section 4.2. For instance, consider a query like $Q(x, y) :- Flight(x, y), Flight(y, x)$ which is searching for pair of cities that round-trip flight is possible. A rewritten query in GAV would be in the terms of data sources since the goal is to answer the query by using only the data sources. Thus, Q can be re-written by the followings:

- $Q'_1(x, y) : \neg WestJet(x, y), WestJet(y, x)$
- $Q'_2(x, y) : \neg WestJet(x, y), AirCanada(y, x)$
- $Q'_3(x, y) : \neg AirCanada(x, y), WestJet(y, x)$
- $Q'_4(x, y) : \neg AirCanada(x, y), AirCanada(y, x)$

As this example shows a rewriting in GAV. For each subgoal g in the query, if a rule in M exists such that the head predicate symbol of M is the same as g 's predicate symbol, then, first, the variables appearing in the head of M are renamed with respect to the g in such a way that they become identical, and second, g is replaced with

the renamed body of M . Note that the non-distinguished variables in M should be replaced with unique names that have not appeared before.

GAV is not appropriate for the domains in which changes are constantly taking place. For example, assume that a data source is changed; consequently, some modifications might be needed in those mappings in which the Web service is used. If these modifications occur during the rewriting or after the rewriting, then the results cannot be reliable. However, GAV is appropriate for those domains that have fixed data sources and changeable global schema.

4.3.2 Local-As-View (LAV)

LAV-based systems, like any integration system, consist of $\langle G, S, M \rangle$ where G is a datalog syntax, i.e., $G = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, and S is a set of sources' names. The sources are assumed each to have a unique name. In these systems, the set G is defined independently from S , and then the mappings between these two sets are established.

In LAV, in contrast with GAV, the set M includes mappings from S to G .

A mapping in LAV is a conjunctive query over the language $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$ such that the predicate symbol used in the head belongs to the set S , and the body contains \mathcal{L} -atoms. Each rule in M is called the *view* of the data source, that appears in the head of the rule. In the rest of this thesis, we refer to views as the logical descriptions of data sources.

Definition 4.3.1 (view) *Let $\langle G, S, M \rangle$ be an LAV-based system where $G = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$ is a datalog syntax. In this system, a view can be seen as a logical description of a data source, e.g. ds , in S . Views are defined by uniquely named conjunctive*

queries (commonly denoted by V) over the global schema G . The answers of this query (denoted by ν) is the data that is stored in ds . A view is complete if $\nu = V^{I_\nu}$ where V^{I_ν} is the results of interpretation of conjunctive query V over I (see definition 4.2.2); view is incomplete if $\nu \subset V^{I_\nu}$ \square .

Mappings in LAV enable the system to be more flexible to changes in data sources. If a data source is changed, modifying only the mappings in M that contain the data source would be sufficient. This flexibility makes LAV-based systems suitable for the domains that the global schema is fixed, and the sources may change. Nevertheless, the query rewriting process in LAV is highly complicated.

To correctly rewrite a query in LAV, we first need to compare two queries. Two interesting query compressions are query containment and query equivalence. Briefly, query Q_1 is contained in query Q_2 if any answer of Q_1 is also the answer of Q_2 . Two queries are equivalent if the set of answers of Q_1 and Q_2 are the same. The formal definitions of these query compressions are presented in definitions 4.3.3 and 4.3.4.

Definition 4.3.2 (Partial query containment and equivalence) *Given conjunctive queries Q_1 and Q_2 , both described w.r.t. a datalog syntax \mathcal{L} and an Interpretation I for \mathcal{L} , Q_1 is partially contained in Q_2 under I , denoted by $Q_1 \sqsubseteq_I Q_2$, if $Q_1^{I_{Q_1}} \subseteq Q_2^{I_{Q_2}}$. Q_1 and Q_2 are partially equivalent, denoted by $Q_1 \equiv_I Q_2$, if $Q_1 \sqsubseteq_I Q_2$ and $Q_2 \sqsubseteq_I Q_1$ \square .*

Definition 4.3.3 (Query containment) *Given conjunctive queries Q_1 and Q_2 , both described according to a datalog syntax \mathcal{L} , Q_1 is contained in Q_2 if for any interpretation I w.r.t. \mathcal{L} , we have $Q_1 \sqsubseteq_I Q_2$; it is denoted by $Q_1 \sqsubseteq Q_2$.*

Definition 4.3.4 (Query equivalence) *Given conjunctive queries Q_1 and Q_2 , both described according to a datalog syntax \mathcal{L} , Q_1 and Q_2 are equivalent, denoted by $Q_1 \equiv Q_2$, if for any interpretation I w.r.t. \mathcal{L} , we have $Q_1 \equiv_I Q_2$. \square .*

The problem of query rewriting using views in LAV is the following. Suppose we are given a query Q over a global schema, and a set of view definitions V_1, \dots, V_n over the same schema. The question is to check the possibility of answering the query Q using only the answers to the views V_1, \dots, V_n [36]. Now, we have a context to define query rewriting using views as follows:

Definition 4.3.5 (Query rewriting using views) *Given a conjunctive query Q and a set of view definitions V_1, \dots, V_n over the language \mathcal{L} , a rewriting of the query using views is a conjunctive query Q' over the language $\mathcal{L}' = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \cup \{V_1, \dots, V_n\} \rangle$ whose subgoals are the all atoms of $\langle \mathbb{C}, \mathbb{V}, V_1, \dots, V_n \rangle$. \square .*

Example 4.3.1 Consider the following query and views which are defined over the language L in example 4.2.1:

- $Q(x) : \neg Student(x)$
- $V_1(x, y) : \neg SupervisorOf(x, y), Happy(x)$
- $V_2(x) : \neg Happy(x), Student(x)$
- $V_3(y) : \neg SupervisorOf(x, y), Student(x)$

All the queries Q'_1 , Q'_2 , Q'_3 , and Q'_4 are the rewritings of Q by using the views. Note that although Q'_3 does not provide the data as the query request, it is a rewriting according to definition 4.3.5.

- $Q'_1(x) : \neg V_2(x)$

- $Q'_2(x) : -V_1(x, z), V_2(z)$
- $Q'_3(x, y, z) : -V_1(x, x), V_2(y), V_3(z)$
- $Q'_4(x) : -V_1(x, 'john'), V_2(x)$

□

Recalling the purpose of data integration systems, when a query is received, the system attempts to answer by rewriting the query in terms of available data sources. Obviously, a rewritten query is supposed to provide related answers to the original query. For example, when a received query is about departure flights from Toronto, rewriting with data sources that provide arrival flights to London is not interesting. However, as the example 4.3.1 shows, various rewritten queries with possibly different answers (if they are evaluated over the same interpretation I) can be provided. We distinguish two groups of rewritings, equivalent rewritings and maximally-contained rewritings, which finding them is the purpose of most of the proposed approaches in the literature. We later show that the goal of our system is to find the maximally-contained rewriting of a query if exist.

Definition 4.3.6 (Equivalent rewriting) *Let Q be a query and V_1, \dots, V_n be views, all over the same datalog language \mathcal{L} ; and let Q' be a rewriting of Q over \mathcal{L}' (see definition 4.3.5). We say Q' is an equivalent rewriting of Q if we have $Q'^{I'Q'} = Q^{IQ}$ for any \mathcal{L} -interpretation I , the \mathcal{L}' -interpretation I' is obtained as $V_i^{I'} = V_i^{(I)V_i}$ for any view name V_i in \mathcal{L}' . □*

Definition 4.3.7 (Maximally-contained rewriting) *Let Q be a query and V_1, \dots, V_n be views, all over the same datalog language \mathcal{L} ; and let $\{Q'_1, \dots, Q'_m\}$ be a set*

of rewritings of Q ($m \geq 1$) over \mathcal{L}' . We say the union of the answers of contained-rewritten queries is maximally-contained in Q if we have $\bigcup_1^m Q_i'^{I'Q_i} \subseteq Q^{I_Q}$, and there is not another different set of rewritings $\{Q_1'', \dots, Q_k''\}$ in \mathcal{L}' such that $\bigcup_1^m Q_i'^{I'Q_i} \subseteq \bigcup_1^k Q_j''^{I'Q_j''} \subseteq Q^{I_Q}$ for any \mathcal{L} -interpretation I with the same restriction as definition 4.3.6 for I' . \square

Since language \mathcal{L}' contains \mathcal{L} , the query Q can be also described over \mathcal{L}' , but the equivalency (or maximally contained) should be examined by a subset (not all) of all the possible interpretations for \mathcal{L}' . As definition 4.3.6 shows, a restriction should be held for describing the interpretation I' for \mathcal{L}' : selecting only those I' in which the interpretation of views are based on the interpretation of P where $P \subseteq \mathbb{P}$. These interpretations can be created by, first, fixing the meaning (P') for all the $P \subseteq \mathbb{P}$, then, interpreting the views based on these meanings. By relaxing this restriction, two rewritings such that one is contained in another may be specified as a non-contained one by the relaxed definition (see example 4.3.2).

Example 4.3.2 Consider the following query Q and view V over the language L (see example 4.2.1) and a given $D^I = \{John, Sara, Kian\}$. A rewritten query Q' can be described over the language $\mathcal{L}' = \langle \{john, sara\}, \{x, y, z\}, \{Student, SupervisorOf, Happy, V\} \rangle$ as follows:

- $Q(x) : \neg Happy(x)$
- $V(x) : \neg Student(x), Happy(x)$
- $Q'(x) : \neg V(x)$

Intuitively, Q' is contained in Q since the results of Q' contain all the happy persons that are also students; however, by having no restriction for creating an interpreta-

tion for \mathcal{L}' , we can create an interpretation I where $Happy^I = \{Kian\}$, $Student^I = \{Sara, Kian\}$, and $V^{I_V} = \{John\}$ and derive Q' is not contained in Q since $Q'^{I_{Q'}} \not\subseteq Q^{I_Q}$. \square

In spite of reducing the number of possible rewritten queries by finding equivalent (or maximally-contained) ones, the set containing these particular rewritings can be infinite because the number of subgoals in rewritten queries are not bounded. For instance, a view can be infinitely repeated in a rewritten query. Thus, a boundary over the length of rewritten queries is vital in order to make the problem of rewriting tractable in a reasonable time.

In [54], two classes of rewritten queries, locally minimal and globally minimal, were determined. It was also proven that these classes lead us to a bound for the length of rewritten queries which can be found in theorem 4.3.1.

Definition 4.3.8 (locally minimal and globally minimal [54]) *Let Q be the query and V_1, \dots, V_n are the views, all over the same datalog language \mathcal{L} . A rewriting Q' is locally minimal if there is no view V used in Q' such that $(Q' \setminus V) \equiv Q'$. The rewriting Q' is globally minimal if there is no rewriting Q'' over L' such that $Q'' \equiv Q'$ and $|Q''| < |Q'|$ ($|x|$ shows the number of subgoals in query x). \square*

Theorem 4.3.1 *Let's assume query Q and a set of views V_1, \dots, V_n w.r.t. a datalog language $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$ such that there are no comparison predicates ($=, \neq, \leq, \geq$) in \mathbb{P} . If Q' is a locally minimal and equivalent rewriting of Q using the views, then $|Q'| \leq |Q|$. [54]. \square*

Up to this point, the number of possible rewritings is reduced by finding special classes of rewritings (i.e., equivalent rewriting and maximally-contained rewriting).

Moreover, the length of rewritten queries are bounded by defining locally minimal rewritings. In order to make the rewriting process practicable, our attempt will be to find all contained rewritten queries which are locally minimal.

To find whether a rewritten query is contained in the original query, a test is required. Containment mapping [18] provides a necessary and sufficient condition for testing query containment. The definition and the theorem of correctness of this testing is described in definition 4.3.9 and theorem 4.3.2 respectively.

Definition 4.3.9 (containment mapping [18]) *Let Q_1 and Q_2 be conjunctive queries w.r.t. a datalog language $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, and let terms in Q_i be the set of all the variables and constants represented in Q_i . A containment mapping $\tau : \mathbb{C} \cup \mathbb{V} \rightarrow \mathbb{C} \cup \mathbb{V}$ from Q_2 to Q_1 is a mapping from the terms in Q_2 to the terms in Q_1 , such that the mapping is identical on the constants and under this mapping (i) the head of Q_2 becomes corresponding to Q_1 's head, and (ii) each subgoal of Q_2 corresponds to a subgoal in Q_1 . Later, we will use the term cover to indicate the corresponding relations between the subgoals such that subgoal s_i in Q_2 covers the subgoal s_j in Q_1 if s_i corresponds to s_j by a given containment mapping τ . \square*

Theorem 4.3.2 *Let Q_1 and Q_2 be conjunctive queries w.r.t. a datalog language \mathcal{L} . Q_1 is contained in Q_2 if and only if there is a containment mapping from Q_2 to Q_1 [18]. \square*

All the preliminaries required to rewrite queries in LAV are reviewed. Since many algorithms with different perspectives have been proposed to rewrite queries in LAV, they are provided in as a separate Section. In the next Section, these algorithms are

classified based on their mechanism, and a comparison is provided as the conclusion of this Chapter.

4.4 Query Rewriting Algorithms in LAV

Various algorithms have been proposed to tackle the problem of query rewriting in LAV-based systems. These algorithms can be classified based on their mechanisms into categories of (i) logical-based algorithms, (ii) bucket-based algorithms, (iii) hybrid algorithms which are the combination of logical and bucket based procedures, and (iv) graph based algorithms.

4.4.1 Logical-based algorithms

Logical-based algorithms are known as algorithms in which purely logical techniques are used to rewrite queries. One of the famous well-studied algorithms in this category is the inverse-rule algorithm [25]. The key step in this algorithm is the inversion of views' descriptions (conjunctive queries) for all views to form new rules. These rules are constructed such that the view's subgoals appear in the head of the new rules, and the body contains the view's head (i.e., view name) and possibly some function symbols. The function symbols are the results of Skolemization. The Skolemization is a way of removing existential quantifiers and existentially quantified variables. For more information about Skolemization and unification in First-Order Logic (FOL), readers are referred to Chapter 18 of [9].

The inverted rules are then used to rewrite the query; each subgoal s in the query is replaced by a body of an inverted rule in which the head is identical to s . These

replacements result in a rewritten query in terms of views (same perspective as query rewriting in GAV described in Section 4.3.1). The result of replacements is then simplified by removing function symbols to form the final query plan.

In detail, the algorithm starts by inversing the rules of views. Each local variable of the view is replaced by a unique function symbol f (the Skolemization constant). For instance, back to our previous examples, the variable z is a local variable of view V .

$$V(x, y) :- \text{SupervisorOf}(x, z), \text{SupervisorOf}(z, y)$$

As mentioned in Section 4.1 and 4.2, rule V can be viewed in the form of

$$\forall x \forall y \forall z. (\text{SupervisorOf}(x, z) \wedge \text{SupervisorOf}(z, y) \rightarrow V(x, y))$$

which is a FOL sentence, and it is logically equivalent to the following FOL sentences:

$$\forall x \forall y \forall z. (\neg \text{SupervisorOf}(x, z) \vee \neg \text{SupervisorOf}(z, y) \vee V(x, y))$$

$$\forall x \forall y \forall z. \neg (\text{SupervisorOf}(x, z) \wedge \text{SupervisorOf}(z, y) \wedge \neg V(x, y))$$

$$\forall x \forall y. \neg \exists z (\text{SupervisorOf}(x, z) \wedge \text{SupervisorOf}(z, y) \wedge \neg V(x, y))$$

By the Skolemization method, the existential quantifier can be eliminated. The function symbol f can be replaced as follows:

$$\forall x \forall y. \neg (((\text{SupervisorOf}(x, f(x, y)) \wedge \text{SupervisorOf}(f(x, y), y)) \wedge \neg V(x, y)))$$

which is logically equivalent to:

$$\forall x \forall y. (((\text{SupervisorOf}(x, f(x, y)) \wedge \text{SupervisorOf}(f(x, y), y)) \rightarrow V(x, y)))$$

The above sentence can be expressed in the following form:

$$V(x, y) :- \text{SupervisorOf}(x, f(x, y)), \text{SupervisorOf}(f(x, y), y)$$

By the Skolemization method, we can eliminate local variables of any views. In the next step, each Skolemized view V is replaced by a set of rules such that the number of rules in the set is equal to the number of subgoals of the Skolemized V . These new rules have the form of (i) one subgoal of V as the head, and (ii) the head of V as the only subgoal of their bodies.

For instance, the Skolemized view V is replaced by the following rules:

$$\text{SupervisorOf}(x, f(x,y)) \rightarrow V(x, y)$$

$$\text{SupervisorOf}(f(x,y), x) \rightarrow V(x, y)$$

To rewrite a query, we can use the modified version of semi-naive algorithm for the GAV approach, which is described in Section 4.3.1. Modifications are required because there is no support for function symbols in this algorithm. Duschka's approach [25] moves the function symbols out of the semi-naive evaluation and put them into a rule-rewriting step. In effect, because there is no nested function symbols, the function symbols always combine with predicates.

The unification technique then unifies two subgoals by finding the simplest substitution for the variables to make them identical. For instance, the $\text{SupervisorOf}(f(x, y), y)$ and $\text{SupervisorOf}(a, c)$ have the unification $\text{SupervisorOf}(f(c, c), c)$ by mapping $x \rightarrow c$, $y \rightarrow c$, and $a \rightarrow f(c, c)$. But the predicates $\text{SupervisorOf}(x, f(x, y))$ and $\text{SupervisorOf}(f(x, y), y)$ have no unification. The unification can be answered in linear time. Thus, it can be added to the GAV-based algorithm (see Section 4.3.1) and the modified algorithm can be used for rewriting in LAV.

4.4.2 Bucket-based algorithms

A naive algorithm for rewriting a query using views is to (i) choose an arbitrary subset of views and (ii) checks whether the conjuncts of chosen views are contained in the initial query by providing containment mapping test. Although the procedure is simple, the algorithm is too expensive, especially for the problems with thousands of views.

By the theorem 4.3.1, we can reduce the search space and set a bound on the number of chosen views. Recalling the locally minimal rewriting (definition 4.3.8), a rewriting Q' is locally minimal if there is no view V used in Q' such that $(Q' \setminus V) \equiv Q'$. Thus, we can bound the size of chosen views based on the size of the query.

Moreover, we can further reduce the search space by eliminating the views that are irrelevant to the query. For instance, when a query is about all the mothers who have more than two children, considering a view which provides a list of flights between two cities is obviously irrelevant. Since there is no ontological structure about the concepts at this point, two concepts are relevant if, and only if, they are identical; otherwise, they are irrelevant. Finding relevant views for rewriting is the intuition behind the Bucket algorithm.

The concept of bucket was proposed in [55] for the first time. The Bucket algorithm considers each subgoal of the query in isolation and creates a new bucket for each subgoal. A view V is added to a bucket B_g (associated bucket to the subgoal g in Q) if the following conditions hold:

- (C1. view V has a subgoal s which has the same predicate as g
- (C2. if there is a distinguished variable in g , then there should be a distinguished

variable in s at the same position (in other word, distinguished variables in g should be mapped to the distinguished variables in s)

The intuition behind the condition C1 is clear; it guarantees that the used views are relevant to Q . C2 is a necessary condition for having a containment mapping from Q to Q' . If C2 is violated, then there would not be any containment mapping (see definition 4.3.9).

After filling the buckets, the algorithm considers all the possible combinations of rewritings. Each combination is constructed by choosing an element (i.e. view) from each bucket, and then providing a containment mapping from the query to these views. If such a mapping exists, then the combination is one of the answers.

The Bucket algorithm, because of removing irrelevant views for rewriting a query, will have a better performance in comparison with the naive algorithm described at the beginning of this Section. In the worst case scenario, when all the views have the same predicate as the query, the performance would be the same as the naive algorithm.

In the Bucket algorithm, the second phase, which is the creation and the verification of combinations, is considerably time consuming. To speed up the rewriting process, we need to reduce the number of possible combinations in the second phase. Fortunately, some of the combinations (e.g. see example 4.4.1) can be eliminated without requiring any extra verification. These combinations are created because query subgoals are investigated in isolation in the first phase of the Bucket algorithm.

Example 4.4.1 *Let Q be the query and V_1 and V_2 be the only views that we have.*

$$- Q(x, y) : -P(x, z), R(z, y)$$

- $V_1(w) : -P(w, e1)$
- $V_2(z) : -R(e2, z)$

Then the bucket for each subgoal of the query would be as $B_P = \{V_1\}$ and $B_R = \{V_2\}$.

The bucket algorithm then chooses one view from each bucket and combines them.

Since no containment mapping exists, the combination will be dropped. \square

By considering query subgoals in isolation we lose the information about what subgoals are joints, i.e, they have shared variable(s). This information can be gained by considering the shared variables which is the intuition behind the Shared-Variable Bucket (SVB) algorithm [66].

4.4.2.1 Shared-Variable Bucket algorithm

The Shared-Variable Bucket (SVB) algorithm [66] is a modified version of the Bucket algorithm. In this algorithm, the as in the Bucket algorithm, a bucket is created for each subgoal of the query. In contrast to the former, a bucket is also created for each shared variable in the query. This bucket contains only views that cover all the subgoals in which the shared variable appears. Moreover, some additional conditions are used for filling the buckets. Before describing these conditions, some concepts are needed to be mentioned.

Recalling from Section 4.2, a variable that appears in the head of a conjunctive query is called *distinguished*, and others are *non-distinguished (existential)* variables. When a subgoal in a rewritten query (i.e. a view's head) is expanded (that means the head is replaced by the body of the view), the non-distinguished variables of the view become *local*. A *local* variable should not appear anywhere else in the rewritten query;

therefore, local variables are renamed to a unique name during expansion. Variables which are not local are called *exposed* variables. These variables are renamed based on the mappings provided during the filling of the buckets as in the Bucket algorithm. The exposed variables are the only ones that may appear in subgoals belonging to the expansion of two or more different subgoals of a rewritten query. The last definition is about shared or unique variables: a query variable is *shared* if it appears more than once in different subgoals; otherwise it is unique. For example, assume a view V and a rewritten query Q' as follows:

- $V(x, y) : -P(x, z), Q(z, y)$
- $Q'(u, v) : -..., V(u, w), ...$

The variables x and y are distinguished variables of the view V , and the variable z is a local one. By expanding the solution we would have:

- $\text{Exp}(Q'(u, v)) : -..., P(u, z), Q(z, w), ...$

As the above expansion shows, the distinguished variable x in V is substituted with u , as well as the substitution of distinguished variable y in V with w . So, u and w in the expansion are exposed variables. Also, u and v are distinguished variables of $\text{Exp}(Q')$. Moreover, u is an exposed, distinguished variable of $\text{Exp}(Q')$, and w is an exposed, non-distinguished variable of $\text{Exp}(Q')$. Note that since z in V is local, it may not appear in any other places in the expansion except those shown.

Back to determining conditions for filling the single buckets in SVB algorithm. The following conditions should be met during mapping of the variables of a subgoal of a query to a subgoal of a view:

- C1. A distinguished query variable can only map to a distinguished variable of view.
- C2. A non-distinguished, shared query variable should not be mapped to a non-distinguished view variable.

For filling the shared variable buckets, view V is inserted to a query shared-variable y only if for any subgoal s in the query in which y appears,

- $C'1$. there exists a subgoal v in V that covers s
- $C'2$. for each non-distinguished shared query variable y' that appears in s and is mapped to a non-distinguished variable of V , V covers all the query subgoals that y' occurs in.

Members of the bucket for a shared variable a is a set of pairs such that the first element is a view head V , and the second element is a set of subgoals v in V such that there is a mapping from all the query subgoals containing a to s . In this mapping, conditions $C'1$ and $C'2$ are met.

As an example to show how the entire algorithm works, assume we have two views such that the first one provides flights with one stop, and the second web service provides flights with two stops. Now, we are looking for flights with five stops. For simplifying, assume the predicate `flight` gets two cities as the input and checks whether they are connected by a flight. Let Q be the query, and V and W be the views that we have.

- $Q(a, b) :- \text{Flight}(a, c), \text{Flight}(c, d), \text{Flight}(d, e), \text{Flight}(e, f), \text{Flight}(f, g), \text{Flight}(g, b)$
- $V(x, y) :- \text{Flight}(x, z), \text{Flight}(z, y)$
- $W(u, v) :- \text{Flight}(u, s), \text{Flight}(s, t), \text{Flight}(t, v)$

The algorithm starts to create a bucket for each query subgoal, e.g., $Flight(a, c)$ by checking all the views. The view V has subgoal $Flight$; therefore, the mapping between them should be checked. The condition C1 is met because a is a distinguished variable of Q and x is a distinguished variable of view V . But condition C2 does not hold because c is a shared variable while z is a non-distinguished one. Hence, the subgoal $Flight(x, z)$ cannot cover the subgoal $Flight(a, c)$ in the query. The algorithm continues by checking the second subgoal of the V , but it cannot cover the subgoal due to the violation of the second condition. Algorithm continues by checking view W , and again the same situation happens. Consequently, the bucket for the subgoal $Flight(a, c)$ would be empty. By repeating this procedure for the other query subgoals, empty buckets would be the result.

In the second part, the algorithm tries to create a bucket for each shared variable of Q , and start with c and the view V . There is a mapping from query subgoals $Flight(a, c)$ and $Flight(c, d)$ (all the subgoals in the query that contain c) to subgoals $Flight(x, z)$ and $Flight(z, y)$ in V since all the distinguished variable in these two query subgoals are mapped to the distinguished variable x . Hence, the pair $\langle V, Flight(x, z), Flight(z, y) \rangle$ is added to the bucket of shared variable c . By checking the view W , the pair $\langle W, Flight(u, s), Flight(s, t) \rangle$ is added to the bucket of c . The bucket for d would be $\{ \langle V, Flight(x, z), Flight(z, y) \rangle, \langle W, Flight(u, s), Flight(s, t) \rangle, \langle W, Flight(s, t), Flight(t, v) \rangle \}$. By continuing this procedure, the bucket for e and f would be the same as d , and the bucket for g would be the same as a .

In the final step, the algorithm provides the Cartesian product between the buckets and then performs the containment testing to verify the results. Since the buckets for the subgoals are empty, we must group the subgoal according to their shared

variables, and cover them, in groups, from the buckets for the shared variables. One of the options for covering the $Flight(a, c)$, $Flight(c, d)$ in the query is the pair $\langle V, Flight(x, z), Flight(z, y) \rangle$ from the bucket c . The options for covering the $Flight(c, d)$, $Flight(d, e)$ of the query are all the three members of the bucket d . But, if we choose W , we would not have any views to cover only the last subgoal in the query. Based on this limitation, there are two solutions:

- $S_1(a, b) : -V(a, m), V(m, n), V(n, b)$
- $S_2(a, b) : -W(a, m), W(m, b)$

As this example shows, instead of providing the $5 \times 5 \times 5 \times 5 \times 5 \times 5$ (15625) containment mapping test, we tested only $2 \times 3 \times 3 \times 3 \times 2$ (108) rewritings.

However, SVB first provides a mapping from g in Q to s in V to fill the bucket associated to g , and then, after the combination of the buckets, it again provides a new mapping from Q to a combination. This mapping is redundant and can be eliminated if we consider the mappings in the previous phase. The main idea underlying the MiniCon algorithm [74] is to use the information gained during the consideration of views to combine views and avoid duplicate mappings.

To reach this point, MiniCon considers the shared variables by setting some rules for mapping which will be described in Section 4.4.2.2; therefore, incorrect combinations shown in the example 4.4.1 will be eliminated without checking the containment mapping. Moreover, MiniCon finds the minimal subset of subgoals in V that satisfies these rules. This avoids redundant mappings; hence, no containment mapping is required after the combination.

4.4.2.2 MiniCon algorithm

Given a query Q and a set of views, the MiniCon algorithm provides a set of rewritings Q' such that the union of answers of these rewritings is maximally contained in Q .

The core of this algorithm is the MiniCon Description (MCD). For each given view, we may have several MCDs. Each MCD has information about the part(s) of a view that can be used to cover some subgoal(s) of Q , i.e. it contains a mapping from a subset of query variables to some variables of a view. Intuitively, each MCD is a fragment of a containment mapping from the query to a rewritten query. Therefore, combining the MCDs means joining the mappings in order to create a complete containment mapping.

Before formally defining MCDs, some terms are needed. First, function $\text{Var}(x)$ is defined to return the set of variables which appear in x . Second, a view subgoal v covers a query subgoal s if there is a mapping τ from $\text{Vars}(Q)$ to $\text{Var}(V)$ such that $\tau(s) = v$. Third, a head homomorphism h on a view V is a mapping from $\text{Vars}(V)$ to $\text{Vars}(V)$ which is defined such that it is the identity mapping on the existential variables, but may equate distinguished variables. This means for every distinguished variable x , $h(x)$ is distinguished, and $h(x) = h(h(x))$.

From [74], an MCD C for a query Q over a view V is a tuple of the form $\langle h_C, V(\bar{Y})_C, \varphi_C, G_C \rangle$ where

- h_C is a head homomorphism on V ,
- $V(\bar{Y})_C$ is the result of applying h_C to V i.e., $\bar{Y} = h_C(\bar{A})$, where \bar{A} are the head variables of V ,
- φ_C is a partial mapping from $\text{Vars}(Q)$ to $h_C(\text{Vars}(V))$,

- G_C is a subset of the subgoals in Q which are covered by some subgoal in $h_C(V)$ using the mapping φ_C (note that not all such subgoals are necessarily included in G_C).

As mentioned before, the main idea in this algorithm is to collect information from shared variables in order to reduce the number of possible combinations in the final step (choosing a view from each bucket and combining them). Therefore, the algorithm creates MCDs that satisfy the following property:

Property 1: Let C be a MCD for Q over V . The MiniCon algorithm considers C only if it satisfies the following conditions.

- C1. each distinguished variable in the query must be mapped to a distinguished variable of the view.
- C2. If an existential variable x_1 in the query is mapped to an existential variable of the view V , then for all the subgoals in the query that include x_1 , there should be some subgoals in the V that cover them.

The intuition behind C1, which comes from the Bucket algorithm, is that since a distinguished variable appears in the results, it cannot be mapped to an existential variable that will not appear in the results. The intuition behind C2 is interesting. First, assume that an existential variable of the query is mapped to a distinguished variable. For instance, consider the following example:

- $Q(x) : -R_1(x), R_2(x, z), R_3(z)$
- $V_1(k, y) : -R_1(k), R_2(k, y)$
- $V_2(m) : -R_3(m)$

To check whether R_2 from V_1 can cover R_2 in the query, we should find a mapping between them which is $x \rightarrow k$, $z \rightarrow y$. No subgoal exists in V_1 that can cover R_3 . However, because z is mapped to a distinguished variable, we can hope that we may find another view that covers R_3 by mapping z in R_3 to a distinguished variable in that view, and then join it with V_1 . In this example, although V_1 cannot cover R_3 , it can be joined with V_2 for answering the query, and variables y and m should then be renamed with the same name (i.e., $Q(x) : -V_1(x, t), V_2(t)$).

In contrast, when an existential query variable is mapped to an existential variable in a view, there is no possibility to use another view to cover the other subgoals in which the existential query variable appears, and then join these views. Hence, the view should be capable of covering all the other subgoals in which this existential query variable appears due to the disability of joining this view to another view.

The second phase combines the MCDs to create conjunctive rewritten queries. Since the MCDs are carefully created, the combination process is simple and efficient because of eliminating redundant mappings which is mentioned in Section 4.4.2.1. The MCDs which meet the property 2 can be combined; This property is defined as follows:

Property 2: Given a query Q , a set of views \bar{V} , and the set of MCDs for Q over the views in \bar{V} , the only combinations of MCDs that are allowed to be constructed have the form of C_1, \dots, C_l where

C1. $G_{C_1} \cup \dots \cup G_{C_l} = \text{Subgoals}(Q)$, and

C2. for every $i \neq j$, $G_{C_i} \cap G_{C_j} = \emptyset$.

By performing the two phases, creating MCDs and combining MCDs, a set of contained rewritten queries will be constructed. The union of this set is shown to be

maximally-contained in the given query [74]. Moreover, this algorithm is sound and complete, i.e., every produced rewriting is contained, and for any possible contained rewriting Q'_1 , there is a rewriting Q' which is produced by the algorithm such that $Q'_1 \subseteq Q'$ [74].

4.4.3 Hybrid algorithm

MCDSAT [3] algorithm encoded the query rewriting problem in LAV as a propositional theory using Conjunctive Normal Form (CNF) formulas. These formulas were then compiled to another normal form called deterministic, Decomposable Negation Normal Form (d-DNNF) [23]. A sentence in NNF is a rooted, directed acyclic graph (DAG) where

- C1. each leaf node is labeled with true, false, $\neg x$, or x (x is a variable or constant)
- C2. each internal node is labeled with \wedge or \vee .

A decomposable NNF (DNNF) [23] is an NNF Δ if for every conjunction $\alpha = \alpha_1 \wedge \dots \wedge \alpha_n$ appearing in Δ , no atoms are shared between the conjuncts of α , i.e., $atoms(\alpha_i) \cap atoms(\alpha_j) = \emptyset$ for $i \neq j$. A deterministic NNF (d-NNF) [23] is an NNF Δ if for every disjunction $\alpha = \alpha_1 \vee \dots \vee \alpha_n$ appearing in Δ , every pair of disjuncts in α is logically inconsistent that is $\alpha_i \vee \alpha_j \models false$ for $i \neq j$. For example, $(A \wedge B) \vee (\neg A \wedge C)$ is a d-NNF.

Decomposability property permits the complex operations in polynomial-time such as satisfiability, to find an assignment that makes a formula true; projection, to compute the strongest sentence entailed by the theory; and model enumeration, to find all the possible assignments that satisfy all the formulas in the theory [22].

After compiling CNF to d-DNNF, MCDSAT utilized the state of the art SAT solvers to find all possible models of the theory (model enumeration). To reach this point, this algorithm encoded the MiniCon algorithm using propositional language; therefore, it can be seen as a hybrid algorithm. The reason that enhanced the performance of MCDSAT compared with MiniCon, is d-DNNF and its polynomial computational characteristics.

4.4.4 Graph-based Algorithm

In [48], authors proposed GQR algorithm a graph-based approach to rewrite queries in LAV. In this approach, queries and views are defined as graphs; Predicates and their arguments correspond to graph nodes. Predicate nodes are labeled with the name of predicates, and they are connected through edges to their arguments. Therefore, shared variables between atoms result in shared variable nodes. Edges are labeled with respect to the position of variable in the argument of the predicate. To distinct distinguished variables from others, \otimes and \bigcirc are used to depict the distinguished and non-distinguished variables, respectively.

This algorithm first decomposes the query and views to simple atomic subgoals and depict them as graphs mentioned above. The decomposition process of views can be done offline in order to speed up system's online performance. Authors proposed a novel query reformulation phase to maps query graph patterns to views' graphs. The algorithm incrementally build up the rewriting as the graphs are combined. This algorithm maximally rewrites queries and is sound and complete.

4.4.5 Comparison of the Algorithms

The idea underlying SVB is to reduce the number of elements in the buckets by considering shared variables and determining some additional conditions. However, this algorithm suffers from redundant mapping checks, described in Section 4.4.2.1. On the other hand, MiniCon creates MCDs in such a way to avoid redundant mapping tests in the second phase, the combination of MCDs. Therefore, MiniCon performs better compared to the other algorithms in the bucket-based and logical based groups [74].

Nonetheless, in the worst-case scenario, when all the views subgoals can cover any of query subgoals, the computational complexity of any of these algorithms is the same. The time complexity in this particular case is $O((nmM)^n)$ where n is the number of subgoals in the given query, m is the maximal number of subgoals in views, and M is the number of views. This value comes from the following calculation: since any subgoal of any view can be used to cover a subgoal in the query, each bucket will contain mM elements. In the next phase, which is choosing an element from each of the buckets, since any combination will be a contained rewritten query, it leads to the Cartesian product between the buckets, and each combination requires n checking. Therefore, we need $(nmM)^n$ checkings in total.

In general, the performance of MiniCon is shown to be better in practice compared to the inverse-rule algorithm [74]. The difference between these two algorithms is in their second phase. In this phase, the inverse-rule algorithm tries to find a unification for the heads of considering inverse rules that are potential candidate for constructing a rewriting. MiniCon is searching for a set of MCDs that are pairwise disjoint and

cover all the subgoals of the query. Unification is over the variables of the query, while finding MCDs depends to the number of query subgoals. Hence, MiniCon searches in a smaller set. Moreover, MiniCon removes the unusable views that violates the property 1 in the first phase, while these views cannot be detected in the first phase of inverse-rule algorithm.

GQR algorithm likewise the bucket based algorithms, investigate subgoals one by one, while similar to MiniCon or MCDSAT, considers the shared variables to prune the incorrect cases. In contrast with these two algorithms, as they mentioned, it doesn't try to a priori map fragments of query to the views. This mappings comes out naturally as GQR algorithm combine atomic view subgoals to larger ones. Consequently, the second phase of the algorithm needs to combine fewer views.

MiniCon directly provides information about which part of a rewritten query is used to cover a considering query subgoal. Therefore, this information can be used to not only evaluate the preferences more accurately, but also they can assist us in developing a more expressive language for describing preferences. This information plays an essential role in our approach for describing and evaluating user preferences such as preferences outlined in Section 3.3. Therefore, we focus on MiniCon as the core of our system.

4.5 Query Rewriting with Dependencies

When designing the global schema for a specified domain, we often have additional information about the domain that cannot be captured by the terms in the global schema. For example, in a global schema containing the terms *Mother* and *Parent*,

we may need to encode more information by saying *a mother is always a parent*. This kind of information can be encoded by specifying dependencies between the terms of global schema.

Knowledge expressed by dependencies can also be used during the rewriting process to produce additional contained results, which are contained only in the presence of these dependencies. For instance, suppose that a query is about finding all parents, and we have a view which provides a list of mothers, assuming that mothers are already defined as parents, we then can use this view to answer the query.

Query rewriting in the presence of dependencies¹ in LAV was first discussed in [26], where a dependency is defined by a negated expression $\neg\Phi$ where Φ is a positive conjunction of atoms, and all the variables in Φ are universally quantified (e.g., $\neg(Mother(x) \wedge Father(x))$ to show Mother and Father are disjoint). Recently, this problem has become one of the major problems in data integration. Especially in the last few years that semantic Web and Web-based ontology languages such as OWL [20] are introduced.

A dependency can be formalized by using logical languages such as propositional logic, description logic, and fragments of first-order logic. The expressivity of describing dependencies and the complexity of rewriting in the presence of dependencies totally depend on the language we choose. Various dependencies such as functional dependencies, inclusion, full dependencies, tuple-generating dependencies (tgd), and equality-generating dependencies (egd) have been defined. However, only a few of them are used in query rewriting problem such as functional dependency and inclusion [35, 6]. In [10, 15], it is noted that using general tgd's makes the query rewriting

¹This problem is also known as *query rewriting with integrity constraints*.

problem undecidable. Thus, various simplified tgds such as weakly-acyclic tgds and guarded tgds are introduced [14]. To cease this Section, a brief description of dependencies is provided in Table 4.1.

Name	FO-formula	Description
Func. Dep.	$\forall x \forall y \forall z \forall y' \forall z' [A(x, y, z) \wedge A(x, y', z') \rightarrow y = y']$	A is an atom
inclusion	$\forall \bar{x} \forall \bar{y} R(\bar{x}, \bar{y}) \rightarrow \exists \bar{z} S(\bar{x}, \bar{z})$	R and S are atoms. \bar{x} , \bar{y} , and \bar{z} are tuples of terms.
Full Dep.	$\forall \bar{x} (\Phi(\bar{x}) \rightarrow A(\bar{y}))$	$\Phi(\bar{x})$ is conjunction of atoms, and $A(\bar{y})$ is an atom. All variables in \bar{y} are also in \bar{x} .
tgd	$\forall \bar{x} (\Phi(\bar{x}) \rightarrow \exists \bar{y} \Psi(\bar{x}, \bar{y}))$	$\Phi(\bar{x})$ and $\Psi(\bar{x}, \bar{y})$ are conjunctions of atoms. \bar{x} and \bar{y} are the tuples of terms. All the variables in \bar{x} must appear in $\Phi(\bar{x})$.
egd	$\forall \bar{x} (\Phi(\bar{x}) \rightarrow x_1 = x_2)$	$\Phi(\bar{x})$ is a conjunction of atoms. x_1 and x_2 must appear in \bar{x} , and all the variables in \bar{x} must appear in $\Phi(\bar{x})$.

Table 4.1: Description of dependencies

Later in Chapter 6, we extend the MiniCon algorithm to handle full dependencies. In addition, we propose an extension to rewrite queries in the presence of full dependencies which can extend any query rewriting algorithm described in Section 4.4. However the optimized versions of the extension are integrated with MiniCon.

Chapter 5

Related Work

In this thesis, our goal is to propose a query rewriting algorithm adoptable for Web service composition problem and then to create a formal ranking system to rank the composed Web services based on user preferences. Therefore, the works related to this thesis can be classified into two main categories of *query rewriting in the presence of dependencies*, and *Web service composition in the presence of user preferences*.

According to the type of user preferences, the approaches of Web service composition in the presence of user preferences can also be classified into two sub-categories. The first category includes the works that handle user preferences for functional characteristics of Web services, while the second category includes the ones in which the users preferences are related to the non-functional aspects of Web services.

Following this categorization, this Chapter first describes related works for preference-based composition of Web services. Then, since our focus is on user preferences regarding the functional characteristics of Web services, the approaches related to this category of research are reviewed and evaluated by using the example defined in Chap-

ter 3. Finally, the related works for query rewriting in the presence of dependencies are reviewed.

5.1 Web Service Composition

Agarwal et.al [2] proposed a framework to let users describe their preferences on non-functionalities of their desired Web services. In this framework, Web services and the domain ontology are described using standard description logic [5], and preferences are described using fuzzy rules. A set of composition plans are provided, namely *sequence*, *parallel*, *choice*, and *loop*. These composition plans are defined by description logic. For instance, $Loop \sqsubseteq WS \sqcap \exists ws.WS \sqcap \exists times.\mathbb{N}$ where *ws* and *times* are functional roles. This definition implies that an instance of Loop can be related to only one instance of WS via the relation *ws* and to only one natural number via the relation *times*. In this system, users are required to first manually choose one of the plans for composition. The system then automatically selects appropriate Web services for the composition based on the chosen plan.

Several non-functionalities of Web services are taken into account in this work such as the availability rate, response time, and price amount. For each non-functional characteristic, an aggregation function is defined to calculate the value of the non-functional characteristic for the composed Web service with respect to the plan that is used. For example, the response time for a composed Web service constructed by a sequence plan is simply the summation of response time values of all the used Web services, while the response time for a parallelly composed Web services is the maximum of the response time values. These functions are modeled within the ontol-

ogy, and are later used to calculate the corresponding values for the non-functional characteristics of the composed Web services.

User preferences are modeled by using fuzzy IF-THEN rules. A fuzzy IF-THEN rule consists of an IF part, which is a combination of terms and a THEN part, which is exactly one term. Each term is in the form of $A = T$ where A is a variable and T is value. A fuzzy function is defined that assigns a value between $[0, 1]$ to the rules based on their degree of fulfillment w.r.t a composition. Users can describe a preference, e.g, *IF ResponseTime = fast and PriceAmount = cheap THEN Rank = high*. Given a set of preference rules, this system first calculates the associated values of composed Web services, and then ranks them based on the evaluation of defined preferences.

In [84], the authors proposed a preference-based system by exploiting Golog (described in Section 2.3.2.1) for composing Web services, and a combination of First Order Logic (FOL) and temporal logic to describe user preferences about functional and nonfunctional characteristics of their desired Web services. Situation calculus and FOL are used to describe functional and nonfunctional characteristics of Web services, and Golog is used to specify composite Web services by using built-in operators such as *occ* and *final*, where *final(f)* states that fluent f holds in the final situation, and *occ(a)* states that action a occurs in the present situation. For more information about situation calculus, readers are referred to Section 2.3.2.1.

User preferences are expressed using FOL combined with additional operators borrowed from temporal logic operators such as *until*, *always*, and *next*. Each preference has a weight between zero to one that is assigned by users. A preference with lower weight is more preferred than a preference with higher weight. When a preference is satisfied by a composition, its weight will be added to the relevant composition's

weight; otherwise, one is added. A composition with lower weight is preferred to a composition with higher one.

For instance, the preferences below can be described for a travel domain. P_1 states that the user prefers economy flights with a Star Alliance carrier, and P_2 states she prefers direct economy flights with Delta airlines, but she prefers P_1 more than P_2 . P_3 expresses that Hilton hotel never is booked.

- P_1 : $(\exists c, w).occ'(BookAir(c.economy, w)[0.8] \wedge member(c, starAlliance)[0.2])$
- P_2 : $occ?(bookAir(delta, economy, direct))[0.5]$
- P_3 : $always(\neg((\exists h).hotelBooked(h) \wedge hilton(h)))[0]$

Moreover, users can specify alternatives for each preference, which means they can group some preferences such that if any of member is satisfied, the group is then satisfied. The order of checking the member of groups is according to their weights; the lower weight is considered first. If the j^{th} preference is satisfied, the associated weight v_j is added to the situation. In a case that none of the members satisfies the considering situation, the maximum weight in the group is added to the situation. Finally, the composition system uses the preferences to generate the most preferred solutions. This framework uses the preferences during the composition in order to prune unlikely possible situations, thus reducing the time of composition process.

In Lin's approach [56], Web services were described by OWL-S (Section 2.3.1), and preferences are expressed by PDDL3 [33]. Preferences are divided into two groups of basic preferences, which are based on FOL and temporal preferences. Temporal preferences consist of some basic preferences and operators of Linear Temporal Logic

(LTL) such as *final*, *at – most – once*, *sometime – after*, and *sometime – before*. In this approach, first, descriptions in OWL-S and PDDL3 are translated into a planning language. Then, SCUP - the proposed algorithm- combines HTN planning with best-first search that uses a heuristic selection mechanism and computes the cost of a state before decomposing it. A better state is determined by evaluating user preferences. Lin showed this approach had substantially better quality in finding the most preferred plan compared to other planning algorithms.

Traverso et al. [85] looked at preferences as the alternative goals of composition that must be satisfied when the main goal could not be reached. Web services described by OWL-S are translated to state transition system. A state transition system provides a sort of operational semantics to OWL-S such as non-deterministic situation handling and partial observation of state of Web services. Preferences are defined in term of goals of composition. They can be described as simple or complex goals. Simple goals are described by using a propositional logic language. For describing complex goal such as “*try to satisfy goal 1, upon failure, do satisfy goal 2*”, they used EaGLE language [51] that has some operators over simple goals. For instance, for propositional formula p as a simple goal, the complex goal g can be defined as $g := p \mid gAndg \mid gTheng \mid gFailg \mid Repeatg \mid DoReachp \mid TryReachp$. They used a planner to create a plan for composition. Easily converting the created plan to an executable one is the advantage of this planner.

In [67], the authors used context logic to deal with Web service composition. In general, context logic is an extension of first-order logic in which sentences are not simply true, but are true within a context i.e. $isTrue(context, formula)$ is true when the first-order formula is true based on the given context as the input. Dynamic

Description Logic (DDL) [19] is used to describe the context. A DDL language consists of TBox, ABox, and ActionBox which is a set of actions, and each action is a binary tuple of pre-conditions and effects. Preferences also are described in terms of DDLs as well as the query and Web services. The proposed framework checks whether a composition of Web services is possible to satisfy the query by using DDL's reasoner. Sohrabi et al. [82] modified PDDL3 and extended it to HTN-based planner to compose Web services based on user preferences and service regulations/policies that can be defined in terms of a Linear Temporal Logic (LTL). The modified version of PDDL3 in this approach has three constructions: *occ(a)* which indicates primitive task *a* occurs in the present state, *initiate(x)* indicates task *x* initiates in current state, and *terminate(x)* shows that *x* terminates in current state. The semantics of the added constructions are discussed in [81]. Each preference, described in terms of modified PDDL3, has a unique name and a cost that is associated with the preference. There is a built-in function in PDDL3 called *is – violated* that takes as input a preference name and returns the number of times that the preference is violated. As an example for describing the preferences, we can consider these two preferences about a travel domain:

- (preference p1 (sometime-after (terminate arrange-trans) (initiate arrange-acc)))
- (preference p2 (imply (and (hasBookedFlight ? Y) (hasAirline ? Y ? X) (member ? X StarAlliance)) (sometime (occ (pay ?Y CIBC)))))

Preference p1 implies the arranging for accommodation should be start after finishing the arrangement for transportation. Preference p2 states that if a flight is booked

with Star alliance carrier, pay using the user's CIBC credit card. Also, it is possible to define a metric function over the preferences such as ($: metricminimize (+(*40(is - violatedp1)) (*20(is - violatedp2))))$) that specifies p1 is twice more important than p2.

In [79], authors considered user preferences over non-functionalities of Web services, and they use Conditional Preference Network (CP-net). CP-net is a framework for representation and reasoning with qualitative preferences. A CP-net is a graphical model that allows the user to describe preference relations as the notions of preferential dependence or preferential independence and conditional preferential dependence or conditional preferential independence. In the context of Web service composition, CP-nets can be used to capture the user preferences among various assignments of a Quality Of Service (QoS) parameter.

In addition, through CP-nets users can also specify conditional preferences on values for a QoS parameter depending on values assigned to other QoS parameters. For instance, a user prefers to have a more secure Web services. Also, it is already defined for the system that e.g. SSLv3 is more secure than SSLv2. Thus, the system will give the higher rank for those composed Web services that use SSLv3. The system can consider more than one QoS parameter (e.g., tractability and performance). In this case, the system draws the induced preference graph of CP-net. This graph is a directed acyclic graph whose nodes are the conjuncts of variables, and there is an edge (o, o') iff o and o' differ only in a single variable. The root of this graph is the highest-level on preferences. Consequently, a composite Web service can be ranked based on the conjunction of variables in the nodes that they fulfill. If the satisfied node is closer to the root, it means that composite Web service is more preferred (see

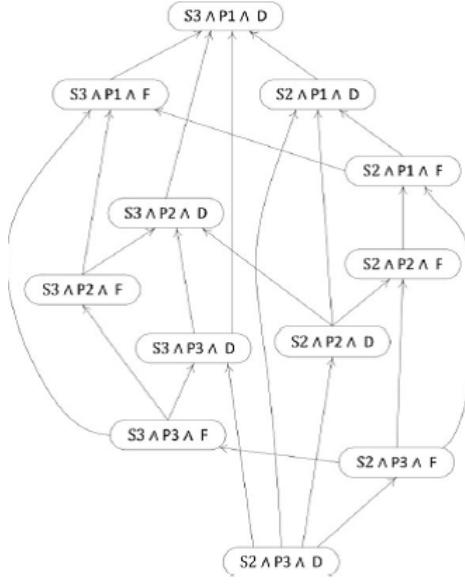


Figure 5.1: Induced Preference Graph [79]

figure 5.1).

Rahmani et al. [75] proposed a heuristic algorithm for Web service composition with considering user non-functional preferences. In this approach, Web services are modeled by their input, output, and their non-functional properties such as execution time and Web service price. The main search algorithm is a backward search. In each step of backward search, there might be more than one relevant candidate service for continuing backward search. Moreover, backward search may try several services which are not reachable from the initial state. By ordering the relevant services, the authors showed that searching can avoid dead end branches.

In backward search, after adding one service to a plan, the service's inputs will be added to the goal list. One simple idea can be about how much criticality one service will add to the problem. The concept of criticality in the proposed algorithm is how far the service's inputs are from the initial state. Hence, they define some functions that

estimate the distance of the input of a service from the initial inputs, and the minimum distance will be chosen. To investigate the non-functionalities of the composed Web service, first, users specify their preferred value for each non-functional property. Then, the same procedure will be operated to those Web services that when they are composed the aggregated distance satisfies the user preferences.

Benouaret's framework [11] can be divided into four sub-components: Data as a Service (DaaS) annotator, RDF query rewriting, Fuzzy constraint matcher, and service ranker. DaaS annotator allows service providers to describe functionalities and (fuzzy) constraints of a service in the form of SPARQL queries. Borrowing from [11], the following SPARQL query illustrates the functionality and constraints of a DaaS service:

```
RDFQuery {SELECT ?y ?z ?t
WHERE{ ?Au rdf:type AutoMaker ?Au name $x
      ?Au makes ?C ?C rdf:type Car ?C hasName ?y
      ?C has Price ?z ?C hasWarranty ?t}}
CONSTRAINTS {?z is 'URL/CheapService', ?t is 'URL/ShortService'}
```

The RDF query rewriter identifies relevant services that match a query or at least some parts of it. The fuzzy constraints matcher is used to compute the matching degrees between (fuzzy) preference constraints and (fuzzy) service constraint for each relevant service. The role of the service ranker is to rank both individual and composite services based on their associated matching degrees.

As an example, a query and preference such as *return the French cars, preferably at an affordable price with a warranty around 18 months and having a normal power with a medium consumption*, can be specified as follows:


```

SELECT ?n ?pr ?w ?pw ?co
WHERE{ ?Au rdf:type AutoMaker ?Au hasCountry 'France' ?Au makes ?C
      ?C rdf:type Car ?C hasName ?n C has Price ?pr
      ?C hasWarranty ?w ?C hasPower ?pw ?C hasConsumption ?co}
PREFERING{?pr is 'URL/AffordablesService',
          ?w is 'URL/around(18)Service', ?pw is 'URL/NormalService',
          ?co is 'URL/mediumService'}

```

In Izquierdo et al approach [40], which is our main motivation, the authors use query rewriting techniques and logical theories to compose Web services. Web services are semantically described using LAV mappings in terms of an ontology domain, and user requests are defined in form of conjunctive queries with respect to the ontology. In addition, users may specify a set of preferences in order to rank the possible solutions to a received request. The LAV formulation allows the system to cast the service selection problem as a query rewriting problem. In this system, the ontology is defined by using subsumption rules. Each Web service description is defined in the form of conjunctive queries over the ontology domain. Each request consists of a conjunctive query and a set of preferences. Preferences are described in form of propositional formulas.

For instance, in the travel domain, users can specify their preferences such as *"if my travel is operated by an airline, I prefer to have a flight provided by AirCanada"* by using the formula like $flight \rightarrow AirCanada$. For each preference a cost can be defined by the user such that when the preference is violated by a composed Web service, its cost will be added to the composite Web service, and finally the composed Web services will be ranked based on their associated costs such that a Web service with a lower cost has a higher rank.

In their proposed mechanism, users can prioritize their preferences by allocating different costs to preferences, e.g. a user wants to imply "*I highly prefer to not fly, but if it is not possible, I prefer to fly by AirCanada*". These preferences can be described as follow: (i) $P1: (\neg flight, 40)$, (ii) $P2: (flight \rightarrow AirCanada, 10)$. Hence, P1 is four times more important than P2. After describing the request, this framework rewrites the query based on available services in the service registry, and then uses a SAT solver in order to evaluate each composite Web service and to rank them based on the evaluated user preferences, and returns a ranked set of composed Web services.

Mesmoudi et al. [65] combined configuration and query rewriting methods for Web service composition. In general, configuration [46] is to find a set of concrete objects that satisfy the properties of a given model. In this approach, services are described in terms of inputs and outputs with respect to the ontology's concepts. The ontology is described by description logic (FLE). This approach uses two algorithms, the first one for identifying services according to the concepts in the query, and the second one for pruning irrelevant rewritings. Then, the configuration method is used to capture the dependencies of selected Web services by drawing the dependency graph which is a directed acyclic graph. Finally, the preferences are specified using the language FLE. The function "*concept score*" is used for each concept in the preference to calculate the degree of relevance between a composition concept and the one concept in P. Next a global score for the composition is calculated from the individual concept scores. The concept score is characterized by two elements closeness and specificity. The closeness parameter shows the semantic distance between the concept in P and its correspondence in composition by using the tree diagram of the ontology. For instance, assume an ontology with this structure: $first - class - flight \sqsubseteq flight \sqsubseteq$

travel, and assume in the composition we have the concept *flight*. Now, if in the preference, there is a concept *flight*, then closeness parameter will take the value '1'. Otherwise, if the concept *first-class-flight* is represented in the preference, it takes the value less than one e.g. '0.8', but it takes another number less than '1' such as '0.3' if the concept *travel* is represented in the preference. The idea of our proximate measurement function, explained by definition 7.2.11, is motivated by the closeness function in this work; nonetheless, the calculation process of these two functions are totally different. Specificity is related to the hierarchical position of the concept in the ontology. A global score can be reached by calculating the average of concept scores for each composition, and they will be ranked based on these numbers.

5.1.1 Evaluation of the Related Works

In this Section, the approaches that considered preferences over the functionalities of Web services are evaluated by our example (see Chapter 3) such as Agrawals approach [2] and Izquierdo et al approach [40]. In other word, the approaches (e.g., [79]) that focus on the users non-functional preferences are out of the scope of this Section.

To evaluate the Izquierdo et al approach [40], the domain ontology should be defined. Predicates *World(x)*, *Europe(x)*, *Scandinavia(x)*, *Africa(x)* are added to the ontology to verify the location of x . *Flight(x,y)* and *train(x,y)* are used to determine that the movement from x to y is done by a flight (or respectively, train).

The Web services in the assumed service registry, which are listed in table 7.3) should be first described. In this approach, the descriptions of Web services are expressed using conjunctive queries in the terms of the concepts in the domain ontology. There-

fore, according to the above ontology, Web service *EFlight* can be described as:

$$EFlight(\$x, y) : \neg Flight(x, y), Europe(x), Europe(y)$$

where *Flight* and *Europe* are relational symbols in the ontology, and predicate *Flight* verifies whether the provided trip is flight. The symbol '\$' denotes that x is input attribute, and the service provides information in the form of a tuple (x, y) .

The descriptions of the other Web services are represented as follows:

- $ETrain(\$x, y) :- Train(x, y), Europe(x), Europe(y)$
- $ScandinaviaTrain(\$x, y) :- Train(x, y), Scandinavia(x), Scandinavia(y)$
- $LocalSwedenAirlines(\$x, y) :- Flight(x, y), Sweden(x), Sweden(y)$
- $WestJet(\$x, y) :- Flight(x, y), World(x), World(y)$
- $EgyptAirlines(\$x, y) :- Flight(x, y), Africa(x), Africa(y)$
- $AirCanada(\$x, y) :- Flight(x, y), World(x), World(y)$
- $EFlight(\$x, y) :- Flight(x, y), Europe(x), Europe(y)$
- $StarHotel(\$x, hotelName) :- World(x), 4-starHotel(hotelName),$
 $LocatedIn(x, hotelName)$
- $AAHotel(\$x, hotelName) :- World(x), 3-starHotel(hotelName),$
 $LocatedIn(x, hotelName)$
- $RoyalHotel(\$x, hotelName) :- World(x), 4-starHotel(hotelName),$
 $LocatedIn(x, hotelName)$

In this framework, a user request R is in the form of tuples $R = \langle Q, P \rangle$ such that Q is a conjunctive query in the terms of the ontology, and P is a set of preferences for Q . Back to our case study, to find the second desired Web service, a query such as the following can be defined:

$$Q(\$x, y) : \neg World(x), Trip(x, u), World(u), Trip(u, y), World(y), Trip(y, w), \\ World(w), Trip(w, x)$$

A preference in this framework is a tuple $\langle \rho, c \rangle$ where ρ is a propositional formula in the terms of Web services names and the ontology's concepts. c is the cost associated with ρ , and will be added to the total cost of a rewritten query when it violates the associated propositional formula. The validity of a preference is defined with respect to the propositional model $M(I)$ such that a valid rewriting I satisfies a preference if either the Web service name appearing in ρ is also appear in I , or a concept (or its parents in the ontology) in the preference appears in I as well. The solution for the request R is any best-ranked valid rewriting based on the total violated costs. By using the language that is proposed in [40], our preference for the above query, which is "*Trips are preferred to be flight*", can be described as $(Flight, k)$ or $(\neg Train, k)$ such that k is the cost of the preference.

Although it is possible to describe this preference in the proposed language, there are some issues in ranking the results. For instance, the following rewritings are valid answers to the request:

- $I1(x, y) : \neg EFlight(x, u), WestJet(u, y), AirCanada(y, w), WestJet(w, x)$
- $I2(x, y) : \neg LocalSwedenAirlines(x, u), ScandinaviaTrain(u, y), \\ ScandinaviaTrain(y, w), ScandinaviaTrain(w, x)$

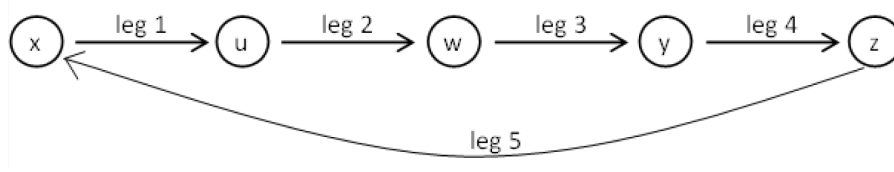


Figure 5.2: Graphical representation of the query

- $I3(x, y) : \neg LocalSwedenAirlines(x, u), ScandinaviaTrain(u, y), AirCanada(y, w), WestJet(w, x)$

For the preference $(Flight, 50)$, the results $I1$, $I2$, and $I3$ have the same ranking since all of them satisfy the preference; thus, the associated total costs are zero. For the preference $(\neg Train, 50)$, the results are ranked as $I1$, $I2$, and $I3$ with the costs of 0, 50, 50, respectively. As this example shows, this approach cannot distinct between the composed Web services that provide three leg flights and the Web services that provide only one leg flight.

For the second desired Web service (visiting four European cites), the query can be specified as $Q(\$x, u, w, y, z) :- World(x), Trip(x, u), Europe(u), Trip(u, w), Europe(w), Trip(w, y), Europe(y), Trip(y, z), Europe(z)$.

The graphical representation of this query is depicted in Figure 5.2.

The preference “If two consecutive cities are in Scandinavia, the travel between them should be done by train” for this query can be expressed by $(Scandinavia \rightarrow Train, 100)$ formula, but this formula does not capture the entire meaning that we expect. For instance, a rewritten query such as $I(\$x, u, w, y, z) :- WestJet(x, u), ETrain(u, w), LocalSwedenAirlines(w, y), EFlight(y, z)$ will satisfy the preference; however, this composition is not the one we prefer because w and y are located in Scandinavia and

connected by a flight.

The preference for the fourth desired Web service cannot be described in this language.

The fifth preference, similar to the first preference, has issues in the ranking process; for example, assume the case that there is no Web service to book a 3-star hotel and there are two Web services such that the first one books a 4-star hotel, and another one books a room in a 5-star hotel. Although, none of these two Web services satisfy the preference, the first one is closer to our preference, and a higher rank is desired to be assigned to it. However, this framework cannot handle this desire.

In [65], the description logic FLE is used for describing the ontology. The syntax and semantics of FLE are represented in Figure 5.3:

<div> <div>□ Concept constructors:</div> <div> $D, E \longrightarrow$ <div> A primitive concept $D \sqcap E$ conjunction $\forall R.D$ universal quantification $\exists R.D$ existential quantification </div> </div> </div>	
<div> <div>□ TBox:</div> <div> $A \sqsubseteq D$ </div> </div>	
<div> <div>□ Abox:</div> <div> $C(a) \quad , \quad R(a, b)$ </div> </div>	
Construct	Set Semantics
$(\forall R.D)^{\mathcal{I}}$	$\{d_1 \in \Delta^{\mathcal{I}} \mid \forall d_2. (d_1, d_2) \in R^{\mathcal{I}} \Rightarrow d_2 \in D^{\mathcal{I}}\}$
$(\exists R.D)^{\mathcal{I}}$	$\{d_1 \in \Delta^{\mathcal{I}} \mid \exists d_2. (d_1, d_2) \in R^{\mathcal{I}} \wedge d_2 \in D^{\mathcal{I}}\}$
$P(f_1, \dots, f_n)^{\mathcal{I}}$	$\{d \in \Delta^{\mathcal{I}} \mid \exists d_1, \dots, d_n \in \Delta^{\mathcal{I}} : f_1^{\mathcal{I}}(d) = d_1, \dots, f_n^{\mathcal{I}}(d) = d_n \text{ and } (d_1, \dots, d_n) \in P^{\mathcal{D}}\}$
$(D \sqcap E)^{\mathcal{I}}$	$D^{\mathcal{I}} \cap E^{\mathcal{I}}$

Figure 5.3: Syntax and semantics of FLE

where A is a primitive concept, C, D, E are arbitrary concepts, R is an arbitrary role, and a and b are individuals. For further detail about description logics, readers are directed to [5].

In this framework, Web services are categorized based on their functionalities. For instance, in our case study, Web services in the registry can be categorized into two main groups: Trip and Accommodation. Each category represents the functionalities of its Web services, and it is described in the terms of the concepts in the ontology. For example, the Trip category is described as $Trip \sqsubseteq \exists HasInput.Location$, and the output *departurePlace* by using the expression $Transportation \sqsubseteq \exists HasOutput.Location$. To indicate the subcategories (e.g., flight and train), the subsumption rule is used; for example, $Flight \sqsubseteq Trip$. Also, a subsumption rule is used to declare the category of each Web services in the repository such as $AirCanada \sqsubseteq Flight$. The names *Flight* and *AirCanada* in the ontology are called abstract Web services. Those abstract Web services that can be replaced immediately by a concrete Web service are called primitive Web services. For our example, AirCanada, WestJet, AAHotel, and ETrain are primitive services. In this approach, only the primitive services are considered: the categories in the ontology that do not have abstract services as subcategories. In other word, the source nodes (i.e. nodes that have no incoming edge) of the ontology graph are considered.

A query in this approach consists of two parts: the mandatory part M and the preference part P . The mandatory part is specified as a triple $\langle I, O, C \rangle$ where I denotes the input data, O denotes the output data, and C denotes service categories. For example, the mandatory part of a query for searching the fifth desired Web service can be specified as:

$$M = Trip \sqcap HasInput.Location \sqcap Hasoutput.Location \sqcap Accomidation \sqcap \\ HasInput.Location \sqcap HasOutput.HotelNames$$

The preference part of the query also is a tuple $\langle I, O, C \rangle$ in which I denotes

the preferences for the inputs, O denotes the preferences for the outputs, and C denotes the category of the desired Web service. For example, the preference “*The booked room is preferred to be in a 3-star hotel*” can be expressed by the expression $\langle ' ', 3starhotel, Accommodation \rangle$. As mentioned in Section 5.1, this approach assigns a value to each concept in the preference to represent the relevance degree of a composition and a concept in P .

This value is called the concept score and is calculated by the formula $S(c, R) = Closeness(c, R) * Specificity(c)$ where R is a composition, c is a concept in the preference. Closeness and specificity are two elements that are defined as follows:

where V is the set of Web services that are used in the composition. The function $extent(c)$ is the semantic extent implied by the concept c . It is related to the hierarchical position of the concept in the ontology, and intuitively measures the granularity of a domain concept. For example, let us take $X \sqsubseteq Y$ and $Y \sqsubseteq Z$, if X has no sub-concepts, the extent value of X is 1, for Y , the extent value is 2 and for Z , the extent value is 3.

Based on these definitions, assume the results of the query described above are compositions of *WestJet* service with *StarHotel*, *AAHotel*, or *RoyalHotel* services. The ranking of the results is represented in Table 5.1. Although the preference is a 3-star hotel, the best score belongs to the Web service that operates bookings in a 5-star hotel.

For the first four desired Web services, we need additional concepts to describe the queries properly. Web services are categorized and each category describes the functionalities of its Web services. Based on our categorization, WJ and EF services have the same functionalities. But, they get different inputs. Therefore, extra categories

Table 5.1: Ranked results of compositions

Composition	Closeness	Closeness	Composition score
WestJet + Royal-Hotel	0.8	1/1	0.8
WestJet + StarHotel	0.8	1/2	0.4
WestJet + AA-Hotel	1	1/3	0.33

are needed. These modifications can be done by eliminating the input part for category *Trip* and describing the inputs of the category in its subcategories individually, adding *Europe-Trip*, *African-Trip* as the subcategories of *Trip*, and specifying the inputs part for the subcategories, e.g., $EuropeTrip \sqsubseteq \exists HasInput.EuropeLocation$ and $Europe-Trip \sqsubseteq \exists Hasoutput.EuropeLocation$. Now, assume that two Web services are added to the repository; the first one operates on Swedish local flights, and the second one only gets French cities as the input. We cannot add them to *Europe-Trip* category due to the different inputs. As such, the initial issue still remains. Consequently, the categories must be divided to very small categories, so in some cases each subcategory has only few Web services, so this issue makes the approach less efficient. On the other hand, if this kind of categorization is done, there are still some issues in describing preferences. For example, describing the third preference *at least three stops in Africa* is not possible due to the structure/language of preferences.

The proposed language for preferences in [84] can describe the first preference as the formula $(always(bookFlight))$, but issues in ranking still remain. For instance, those composed Web services that perform all legs flight itineraries will get the highest rank, and another composed Web services will get the same and a lower rank. In other word,

the results can be divided in two groups: All legs flight and others, and there is no ranking in the second group between e.g. three-leg flights and two-leg flights. The third preference can be described in proposed preference language which is based on FOL, however the ranking is not completely correct. For example, the framework does not make a difference between the group of composed Web services that provide itineraries with two stops in Africa and the other group of services that provides itineraries with no stops in Africa. Also, this approach is not capable of handling the second and fourth preferences because user preferences can only be described for the entire query not a part of it.

The language PDDL3 that is used by [57] and [83] can completely handle the first preference because this language counts the number of times that a preference is violated. Therefore, a composition with three leg flight and one leg train (one time violation) has lower cost than a composition with two leg flight and two leg train (two time violations), and consequently has higher rank. However, there is no possibility for describing preferences like the second and fourth ones.

To summarize, Table 5.2 shows the evaluation of related works.

Table 5.2: Related works evaluation

Approach	Prf1	Prf2	Prf3	Prf4
Izquierdo et al. [40]	-	-	-	-
Mesmoudi et al. [65]	-	-	-	partially
Sohrabi et al. [84]	-	-	partially	-
Lin et al. [57]	X	-	-	-
Sohrabi et al. [83]	X	-	-	-

5.2 Query Rewriting using Views in the Presence of Dependencies

The problem rewriting queries using views in the presence of dependencies, a.k.a rewriting with integrity constraints, was studied first in [26], where a dependency is defined by a negated expression $\neg\Phi$ where Φ is a positive conjunction of atoms, and all the variables in Φ are universally quantified. For example, to show mothers and fathers are disjoint, the expression $\neg(Mother(x) \wedge Father(x))$ can be used.

Other dependencies that have been studied are functional and inclusion dependencies. Gryz [35] proposed an algorithm to equivalently rewrite queries in the presence of inclusion dependencies. This algorithm first chases back the query to form query Q' , and it then tries to find equivalent replacement atoms for each atom in Q' by using the set of inclusion dependencies. The author also provided an extension to also find contained rewritings. In [6], authors extended MiniCon algorithm to rewrite

queries in the presence of inclusion dependencies. The algorithm first chases queries and views and then applies modified MiniCon to rewrite the initial query by using the new query and view subgoals. Duschka et al. [27] extended reverse algorithm to maximally rewrite queries in the presence of dependencies by finding recursive query plans. They also extended the algorithm to support full dependencies. The extension finds recursive query plans that are maximally contained in the initial query. The authors also showed that for the dependencies that are not full, i.e. some existential variables also exist in the query, the algorithm may fail to rewrite the query because the semi-native evaluation of datalog program may not terminate.

In [47], authors dealt with the problem finding maximally contained rewritings in the presence of special class of dependencies called conjunctive inclusion dependencies. The proposed algorithm which is an extension for inverse rule algorithm is sound but not complete.

Another class of dependencies which have been considered are tuple-generating dependencies (tgd's). In [10, 15], it is noted that using general tgd's without any constraints makes the query rewriting problem undecidable because the chasing process may not terminate. Therefore, various subclasses of tgd's are considered such as weakly-acyclic tgd's and guarded tgd's. To more information readers are referred to [14].

Full dependency (see Table 4.1) as a special class of tgd's have been considered. As mentioned above, [27] extended inverse-rule algorithm to rewrite queries in the presence of full dependencies. This algorithm provides recursive rewritten queries. In spite of all these efforts, no extension exists for bucket-based algorithm to handle full dependencies. Therefore, our focus was to extend a bucket-based algorithm which is explained in Chapter 6.

Chapter 6

Query Rewriting in the Presence of Dependencies

Defining dependencies over the global schema may assist us to find more contained rewritten queries, which are contained only in the presence of these dependencies. Various dependencies have been defined such as inclusion, full dependencies, and *tgd*'s (see Section 4.5).

In our approach, we use full dependencies, which are logically equivalent to datalog rules, to describe dependencies. We call the entire package of query language and dependencies, which both are defined over the same language, domain ontology. In this Chapter, the domain ontology is first formalized, and then the concepts used for query rewriting in the presence of ontology are defined. Finally, we explain our algorithm to rewrite queries in the presence of full dependencies.

6.1 Domain Ontology

An ontology is a formal, explicit specification of a shared conceptualization [34]. In this definition, “conceptualization” refers to a model of the world. “Shared” indicates that an ontology captures knowledge which is accepted by a group, and it is meaningful for them. “Explicit” means that the type of concepts in an ontology and the constraints on these concepts are explicitly defined. Finally, “formal” means that the ontology should be machine understandable [20].

Some of the well-studied languages to construct an ontology are description logics (DL), rule-based languages, and some fragments of the combination of the two formers [37]. Various description logics are studied to encode human knowledge, such as SROIQ [38], EL++ [4], and DL-lite [16], that underly the profiles OWL DL, OWL EL, and OWL QL, respectively. Description logics, in general, are powerful tools to capture human knowledge but are inefficient in reasoning performance compared with rule-based languages such as datalog which are less expressive but more efficient in reasoning.

Nonetheless, there exists kinds of knowledge which can be captured by only one of these languages or even by none of them. For instance, in SROIQ, the most expressive DL, it is impossible to assert that a person X who has a brother, sister, mother, and father, then all of them have a complete family [37] because predicates with arity more than two are not allowed. This can be done easily using the datalog rule

$$\begin{aligned} &HasCompleteFamily(x,b,s,f,m) :- HasBrother(x,b), HasSister(x,s), HasFather(x,f), \\ &HasMother(x,m) \end{aligned}$$

On the other hand, datalog rules cannot encode the information that a family with

more than five children is a big family, which is easy to represent in SROIQ. To achieve to a more expressive language the first idea that may come to mind is to combine these two languages because both are fragments of first order logic and can be combined together. However, combining description logics, even EL++ the simplest one, with rules makes the reasoning problem undecidable. Therefore, fragments of this combination such as DL-rules and safe DL-rules have been proposed and studied. For more information about description logics and rules, readers are referred to Chapter 5 and 6 in [37].

In spite of the efforts, having expressivity and tractability is a double-sided sword, and there is a trade-off between these two desires. To handle this trade-off, in our approach, datalog rules are used to construct the domain ontology because in the Web service composition where real-time answering is required, tractability is the main concern. Moreover, we can mix ontological rules with queries and views, and then use the same reasoners for them without any additional cost to translate languages to each other. The formal definition of ontology in this work can be found in Definition 6.1.1.

Definition 6.1.1 (domain ontology) *A domain ontology is a tuple $\mathbb{O} = \langle \mathcal{L}, \Delta \rangle$ where \mathcal{L} is a datalog syntax (Definition 4.1.1) to capture the domain's concepts and their meanings, and Δ is a finite set of safe datalog rules over the \mathcal{L} to capture the relations between these concepts. For any rule in Δ , the order of its subgoals is assumed to be fixed. Moreover, any rule in Δ has a unique id. \mathcal{L} and Δ are called ontology language and ontology rules respectively in the rest of this thesis. \square*

The order of subgoals has no effect on the evaluation of the rules. Hence, without

loss of generality, we can assume that the order of subgoals in each rule is fixed. This characteristic assists us to rewrite the query faster by reducing the number of checks. Before continuing this Chapter, a set of functions are defined. The function $Head(R)$ returns the head atom, while the function $Body(R)$ returns the subgoals of the given rule R ; function $Var(P)$, recalling from Chapter 4, returns all the variables appearing in P ; and $Predicate(P)$ returns the predicate name which is used in P . Moreover we use symbol \models and \vdash to describe consistency/provable. $I \models \Delta$ means for any rule $R \in \Delta$, $R^I = true$. $\Delta \vdash R$ means R is provable based on Δ . In other word, for any interpretation I such that $I \models \Delta$, we have $R^I = true$.

In the rest of this Section, we will define a set of concepts on which our algorithm is based. The first concept that we use during the rewriting is a dependency graph, which is drawn according to Δ and is called Δ -graph. This graph is defined as follows:

Definition 6.1.2 (Δ -graph) *For a given domain ontology $\mathbb{O} = \langle \mathcal{L}, \Delta \rangle$, Δ -graph is a directed acyclic graph $G_\Delta = \langle V, E \rangle$ with labeled nodes and edges such that*

- $V = \mathcal{C} \cup \mathcal{P}$ where \mathcal{P} is a set of nodes labeled with predicate symbols (called *p-node*), and \mathcal{C} is the set including all the nodes labeled with the rules' ids in Δ (called *conjunct-nodes*)
- E is the set of labeled edges that are drawn based on the rules; for a given rule Number n) $P_1(u_1) :- P_2(u_2), \dots, P_k(u_k)$, Δ -graph is drawn as follows:

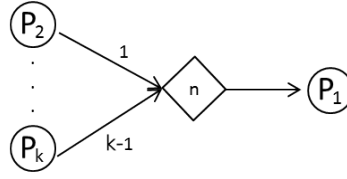
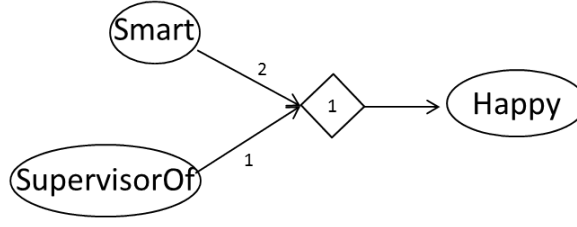


Figure 6.1: Δ -graph basic structure

where the edge labels represent the positions of connected p-nodes in the rule n ; the square nodes represent c-node; and circle nodes represent p-nodes. Note that we only consider the domain ontologies whose their Δ -graph is acyclic. \square

Each predicate in \mathcal{L} has a unique p-node, and each rule in Δ has a unique c-node. As Figure 6.1 shows, there always exists a path from subgoals of a rule to its head, and the length of this path is equal to two. Paths, e.g. from P to S , can be used to check if there exists a rule $\Delta \vdash R$ such that $Head(R) = S$ and P is a subgoal in R . Note that the existence of a path is a necessary but not sufficient condition for finding R . It is not sufficient because predicates' arguments are not considered in Δ graph. We consider them later (see Definitions 6.2.5 and 6.2.6) during the creation of rule R based on some paths in the graph. Therefore, such a rule R may not exist while there is a path from P to S .

Example 6.1.1 Let $\mathcal{O} = \langle \langle \mathbb{C}, \{v_1, v_2, v_3, v_4, x, y\}, \{Smart(v_1), SupervisorOf(v_3, v_4), Happy(v_2)\} \rangle, \{Happy(y) :- SupervisorOf(x, y), Smart(x)\} \rangle$ be a domain ontology; the Δ -graph can be drawn as follows:



□

Definition 6.1.3 (path) *A path from a node labeled with α to another node labeled with β in Δ -graph is a string of nodes that are connected to each other. A path is formalized as a tuple $\langle r_1 : s_1, r_2 : s_2, \dots, r_n : s_n \rangle$ where r_i is an id indicating the rule R_{r_i} in Δ , and s_i indicates the subgoal g_{s_i} in R_{r_i} such that:*

- $Predicate(g_{s_1}) = \alpha$
- $Predicate(Head(R_{r_l})) = Predicate(g_{s_{l+1}})$ for $1 < l < n$
- $Predicate(Head(R_{r_n})) = \beta$

□

Note that since the Δ -graph is acyclic, and Δ is finite, any path in Δ is finite. This characteristic is crucial for query rewriting in the presence of full dependencies, as it guarantee the problem to be decidable. The path concept is used later, during rewriting of a query, to check whether a subgoal of a view has any potential to cover a query's subgoal. To enable such a check, another concept is required, which is path overlap. We can easily determine whether or not two given paths overlap each other by checking their definitions.

Definitions 6.1.4 Path overlap *Two paths $\mathcal{P}_1 = \langle r_1 : s_1, r_2 : s_2, \dots, r_n : s_n \rangle$ and $\mathcal{P}_2 = \langle r'_1 : s'_1, r'_2 : s'_2, \dots, r'_m : s'_m \rangle$ overlap each other if there is an r_i in \mathcal{P}_1 which is*

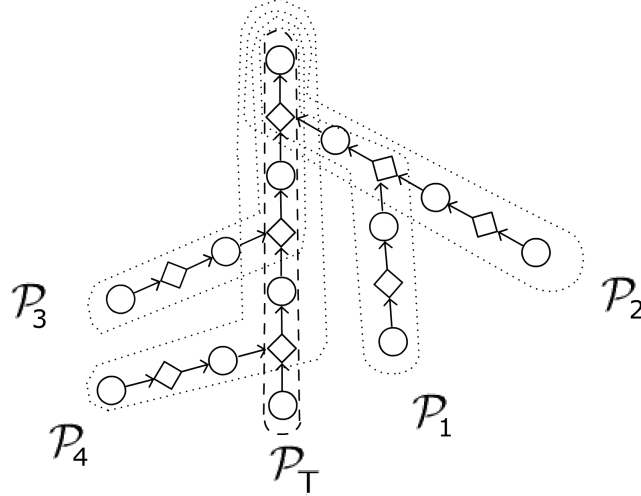
identical to an r'_j in \mathcal{P}_2 . The size of the overlap is equal to the number of r_i 's which are the same in \mathcal{P}_1 and \mathcal{P}_2 . \square

We defined all the concepts Δ -graph, path, and path overlap in order to be able to define a concept called D-RAD, which is described in Definition 6.1.5.

Definitions 6.1.5 D-RAD (Derivation-Rooted Acyclic Digraph) *D-RAD* D is a sub-graph of Δ -graph containing a set of paths $\mathfrak{P} = \{\mathcal{P}_1, \dots, \mathcal{P}_k\}$ such that

- C1)* there is one and only one p -node T_D , called the root, such that for every other node n in D , there is a path from n to T_D , and
- C2)* only one edge goes through each p -node in the *D-RAD*.

Nodes that have no incoming edge are called source node. The path in \mathfrak{P} with maximum overlaps with the other paths in \mathfrak{P} is called the rooted path of the *D-RAD* and is shown by $\mathcal{P}_{\mathsf{T}_D}$. Path $\mathcal{P}_{\mathsf{T}_D}$ may not be unique. In the case that more than one path has maximum overlap, one of them is arbitrarily selected.



\square

The intuition behind the D-RAD is to convert a D-RAD to a new datalog rule R' such that the head of this rule has the predicate T_D , the body contains the source nodes of D-RAD, and the terms are named such that $\Delta \vdash R'$. To pave the way to reach this point, clause C1 is a necessary condition to ensure that the heads of such rules have only one atom. The condition C2 guarantees no non-deterministic situation occurs during the conversion of a D-RAD to a rule.

6.2 Required Tools for using Domain Ontology in Query Rewriting

Domain ontology formalizes the human knowledge about a given domain. This formalized information can be utilized to rewrite a query in order to possibly provide more rewritten queries, which are contained in the initial query only in the presence of the information. This kind of containment is called Δ -containment, and that queries holding Δ -containment are called Δ -contained queries. These queries are formally defined in Definition 6.2.1.

Definitions 6.2.1 (Δ -contained queries) *Let Q_1 and Q_2 be two conjunctive queries, and Δ be the set of ontology rules all w.r.t. ontology language \mathcal{L} . We say Q_1 is Δ -contained in Q_2 , denoted by $Q_1 \sqsubseteq_{\Delta} Q_2$, if for all interpretation I by which $I \models \Delta$, we have $Q_1 \sqsubseteq_I Q_2$*

Theorem 6.2.1 *Assume we have a rule R which is defined as $P_1(u_1) :- P_2(u_2), \dots, P_k(u_k) \in \Delta$ over the ontology language \mathcal{L} . Let $Q_1(\bar{x}) :- P_1(u_1)$ and $Q_2(\bar{x}) :- P_2(u_2), \dots, P_k(u_k)$ be two conjunctive queries over \mathcal{L} where \bar{x} is a tuple of \mathcal{L} -terms; then, Q_2*

is Δ -contained in Q_1 ($Q_2 \sqsubseteq_{\Delta} Q_1$).

Proof. We know that $Q_2 \sqsubseteq_I Q_1$ if $Q_2^{I_{Q_2}} \subseteq Q_1^{I_{Q_1}}$ for all I in \mathcal{L} where $I \models \Delta$. Assume for the sake of contradiction that there is an interpretation $I_1 \models \Delta$ by which $Q_2^{I_{Q_2}} \not\subseteq Q_1^{I_{Q_1}}$, that means, there is an object t in I_1 such that $t \in Q_2^{I_{Q_2}}$ and $t \notin Q_1^{I_{Q_1}}$. Since $t \in Q_2^{I_{Q_2}}$, there is a variable assignment Z_1 such that $P_i^{I_1, Z_1}(u_i) = \text{true}$ for $i = 2, \dots, k$, and $\bar{x}^{I_1, Z_1} = t$. Because all the variables in the head of R must appear in the body (see Definition 4.2.1), Z_1 can also be applied to the terms in u_1 .

On the other hand, $I_1 \models \Delta$ that means for all the variable assignments Z in I_1 , if $P_i^{I_1, Z}(u_i) = \text{true}$ for $i = 2, \dots, k$, then $P_1^{I_1, Z}(u_1) = \text{true}$; therefore, we should have $P_1^{I_1, Z_1}(u_1) = \text{true}$ by the variable assignment Z_1 . Consequently, based on Definition 4.2.2, $(\bar{x}^{I_1, Z_1} = t)$ should be the member of $Q_1^{I_{Q_1}}$ which is in contradiction with our assumption; thus, no such an I_1 exists, i.e., $Q_2 \sqsubseteq_{\Delta} Q_1$. \square

Corollary. For a given ontology $\mathbb{O} = \langle \mathcal{L}, \Delta \rangle$, and the rule $P_1(u_1) :- P_2(u_2), \dots, P_k(u_k) \in \Delta$, let $Q_1(\bar{x}) :- P_1(u_1), \Phi(\bar{y})$ and $Q_2(\bar{x}) :- P_2(u_2), \dots, P_k(u_k), \Phi(\bar{y}), \Psi(\bar{w})$ be two conjunctive queries over \mathcal{L} , where each of $\Phi(\bar{y})$ and $\Psi(\bar{w})$ are conjunctions of \mathcal{L} -atoms that can be empty; and \bar{x} , \bar{y} , and \bar{w} are \mathcal{L} -term tuples. Then $Q_2 \sqsubseteq_{\Delta} Q_1$.

Proof. Since all the ontology rules are safe, all the variables in Q_1 also appear in Q_2 ; therefore, any variable assignment function that can be applied to Q_2 can be applied to Q_1 as well. For any interpretation $I \models \Delta$, If a variable assignment function Z in I satisfies all the subgoals in Q_2 , it will also satisfy all the subgoals in Q_1 , result in $Q_2 \sqsubseteq_{\Delta} Q_1$. \square

As the corollary shows, for two queries Q_1 and Q_2 such that $Q_2 \sqsubseteq_{\Delta} Q_1$, adding some additional subgoals to Q_2 doesn't affect the Δ -containment. We later use this

characteristic to find more Δ -contained queries by using D-RADs and expanding a p-node by its source nodes (i.e. nodes that have no incoming edge).

Example 6.2.1 *By having Theorem 6.2.1 and its corollary, more contained rewritings may be produced. For instance, let assume the ontology $\langle L, \{Happy(y):- Smart(x), SupervisorOf(x,y)\} \rangle$, the query $Q(x):- Happy(x)$, and the view $V(y):- Smart(x), SupervisorOf(x,y)$ all over the same language L . The ontology rule specifies that all the people that supervise smart person(s) are happy, and the view would provide the supervisors of smart people. By having this ontology and using Theorem 6.2.1, we can use the view V to answer the query although the predicate *Happy* does not directly appear in V .*

Definitions 6.2.2 (Δ -equivalent queries) *Two conjunctive queries Q_1 and Q_2 are Δ -equivalent, denoted by $Q_1 \equiv_{\Delta} Q_2$, iff $Q_1 \sqsubseteq_{\Delta} Q_2$ and $Q_2 \sqsubseteq_{\Delta} Q_1$. \square*

For containment testing of conjunctive queries under inclusion and functional dependencies, [41] and [35] used a computation method, called chase. We will adopt this method to check the containment of CQs in the presence of ontology rules (i.e., full dependencies). The chase concept is originally introduced in [60] for functional dependencies. It was modified later in [41] for inclusion and functional dependencies and revisited in [35] for query rewriting in the presence of functional and inclusion dependencies. We modify this method to enable us to utilize ontology rules during rewriting. Using the modified chase method, we can test the Δ -equivalency of queries.

Definitions 6.2.3 ($Chase_{\Delta,R}$) *Let R be an ontology rule $P_1(u_1):- P_2(u_2), \dots, P_k(u_k) \in \Delta$. We say that a $Chase_{\Delta,R}$ is applicable for a conjunctive query Q if the conjunc-*

tion of $P_2(w_2), \dots, P_k(w_k)$ exists in Q and there is a homomorphism h that satisfies the following conditions: (i) constants in u_i are mapped to the identical constants in w_i , and (ii) variables in u_i are mapped to either variables or constants in w_i for $i = 2, \dots, k$. If $\text{Chase}_{\Delta, R}$ is applicable for Q , we then add the atom $P_1(h(u_1))$ to Q to create new CQ Q' . This procedure of adding an atom is called a $\text{Chase}_{\Delta, R}$ step and is denoted by $Q \xrightarrow{h, R} Q'$. \square

Our purpose is to find contained and Δ -contained queries as answers to the initial query by using the MiniCon algorithm because of its performance in practice. However, as described in Section 4.4.2.2, the MiniCon algorithm can only find the contained query as Δ is not supported. Hence, we extend this algorithm to provide Δ -contained queries as well. To reach this point, a set of theorems are required which are defined below.

Theorem 6.2.2 *Given an ontology $\mathbb{O} = \langle \mathcal{L}, \Delta \rangle$ such that $1 \leq |\Delta|$ and a given conjunctive query Q over L , if a CQ $Q^{[1]}$ exists such that $Q^{[1]}$ is the result of a 1-chase $_{\Delta}$ sequence from Q , then $Q^{[1]} \equiv_{\Delta} Q$.*

Proof. $Q^{[1]}$ is applicable, i.e., there is a rule R in Δ and a homomorphism f such that the body of R appears as some subgoals of Q and f maps the terms in R to the query terms which appear in these subgoals. Based on Definition 6.2.3, the homomorphism of the head of R is added as a new subgoal to Q , forming $Q^{[1]}$. Using Definition 6.2.2, we need to show that $Q^{[1]} \sqsubseteq_{\Delta} Q$ and $Q \sqsubseteq_{\Delta} Q^{[1]}$. Showing $Q^{[1]} \sqsubseteq_{\Delta} Q$ is straightforward since all the subgoals in Q are also appeared in $Q^{[1]}$. Therefore, for any variable assignment in any interpretation, if all the subgoals in $Q^{[1]}$ are interpreted as true, all the Q 's subgoals are then interpreted as true, resulting in $Q^{[1]} \sqsubseteq Q$, and then we can

infer $Q^{[1]} \sqsubseteq_{\Delta} Q$. To prove $Q \sqsubseteq_{\Delta} Q^{[1]}$, we only need to investigate the interpretations that are consistent with Δ . Therefore, all the variable assignments in these interpretations that satisfy the subgoals in Q are forced to satisfy the subgoals of $Q^{[1]}$ as well, thus $Q \sqsubseteq_{\Delta} Q^{[1]}$.

□

As mentioned in Definition 4.3.9, a subgoal of a query (e.g., s) is **covered** by a subgoal of another query (e.g., g), if there is a containment mapping from s to g . By having Definition 6.2.3 and Theorem 6.2.2, we can extend the meaning of **covering**. This extension will assist us in constructing a new algorithm, and proving its correctness.

Definitions 6.2.4 (Δ -covering) Let Q_1 and Q_2 be two conjunctive queries, and Δ be a set of ontology rules, all over the same language \mathcal{L} . Then a subgoal, e.g. s , in Q_1 is covered by a subgoal of Q_2 , if either

- there is a containment mapping from s to a subgoal of Q_2 , or
- there is a rule R over \mathcal{L} such that $\Delta \vdash R$, and Q_2 can be chased by R to form $Q_2^{[1]}$ by adding new subgoal g' to Q_2 . Then, there is a containment mapping from s to g' .

□

Recall procedure of rewriting in MiniCon from section 4.4.2.2. For each subgoal g in a query Q , this algorithm compare each subgoal of any view (e.g., s) to g . If the predicate of s is not the same as the predicate of g , then the algorithm concludes that s cannot be used to cover g . By adding the set Δ , the problem of rewriting becomes complicated since we may derive from Δ that s is contained in g . Hence, it is possible to use s to cover g . To handle these situations, we use the Δ -graph.

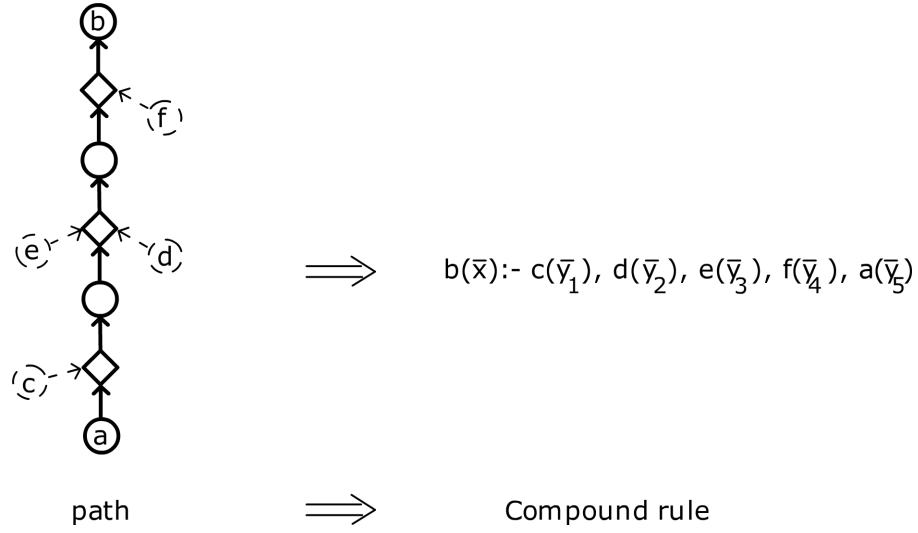


Figure 6.2: Converting a path to C-rule

The main idea underlying our algorithm is to use Δ and check whether s has any potential to cover g when MiniCon fails to use s to cover g . To perform this test, our algorithm attempts to find a path from the node labeled with the predicate of s to another node labeled with the predicate of g . If no such path exists, we can conclude that s has no potential to cover g . The word “potential” is used because even in the case that there is a path from $Predicate(s)$ to $Predicate(g)$, s may not cover g .

Each path in the Δ -graph provides us with a guide to select a subset of rules $\{R\}$ that are required to create a possibly new rule \mathcal{R} which is consistent with Δ (more precisely, $\{R\} \vdash \mathcal{R}$). Intuitively, the body of \mathcal{R} contains the union of all the nodes that are connected to the c-nodes in the path in addition to the first node of the path. The head’s predicate of \mathcal{R} is the last node of the path (see Figure 6.2). The rule \mathcal{R} , which is created based on a path by using a subset of rules, is called Compound rule (C-rule). The procedure of creating a Compound rule from a path is defined in the following definition 6.2.5 and algorithm 1.

Definitions 6.2.5 (Compound Rule (C-rule)) *A compound rule \mathcal{R} is a datalog rule which is created based on a path as a guideline and a set of rules as a source such that \mathcal{R} is consistent with that set of rules. Given a path $\langle r_1 : s_1, r_2 : s_2, \dots, r_n : s_n \rangle$, a compound rule \mathcal{R} is created iteratively by starting from rule r_2 .*

Algorithm 1 Procedure of C-rule creation

- 1: Input: a path \mathcal{P} and a set of ontology rules Δ
 - 2: Output: A compound rule \mathcal{R} if it succeeds; nothing if it fails.
 - 3: Create_C_rule(\mathcal{P}, Δ)
 - 4: create a new rule \mathcal{R} ▷ its head and body will be created in the next steps
 - 5: Find least restrictive head homomorphism h and partial mapping Γ such that

$$\Gamma(g_{s_2}) = h(\text{Head}(R_{r_1}))$$
 - 6: **if** h and Γ exist **then**
 - 7: create function f as follows:

$$f(x) = \begin{cases} \Gamma(x) & \text{if } x \in \Gamma(x) \\ \text{FreshCopy}(x) & \text{Otherwise} \end{cases}$$
 - 8: Body(\mathcal{R}) = $\{f(\text{Body}(R_{r_2}/g_{s_2}))\} \cup h(\text{Body}(R_{r_1}))$
 - 9: Head(\mathcal{R}) = $\{f(\text{Head}(R_{r_2}))\}$
 - 10: **else**
 - 11: terminate and return nothing.
 - 12: **end if**
-

```

13: for each  $3 \leq i \leq n$  do

14:   Find least restrictive head homomorphism  $h_i$  and partial mapping  $\Gamma_i$  such
      that  $\Gamma_i(g_{s_i}) = h_i(\text{Head}(\mathcal{R}))$ 

15:   if  $h_i$  and  $\Gamma_i$  exist then

16:     create function  $f_i$  as follows:


$$f_i(x) = \begin{cases} \Gamma_i(x) & \text{if } x \in \text{Dom}(\Gamma_i) \\ \text{FreshCopy}_i(x) & \text{Otherwise} \end{cases}$$


17:      $\text{Body}(\mathcal{R}) = h_i(\text{Body}(\mathcal{R}))$ 

18:      $\text{Body}(\mathcal{R}) += \{f_i(\text{Body}(R_{r_i}/g_{s_i}))\}$ 

19:      $\text{Head}(\mathcal{R}) = \{f_i(\text{Head}(R_{r_i}))\}$ 

20:   else

21:     terminate and return nothing

22:   end if

23: end for

24: return the created rule  $\mathcal{R}$  as the result

```

Least restrictive head homomorphism h means the minimum number of variables' equations, defined in h , that is necessary to have $\Gamma(g_{s_2}) = h(\text{Head}(R_{r_1}))$.

Algorithm 1 attempts to convert a given path to a rule that can be derived from Δ . Let us consider a simple example: Let $Q(x) : -S(x)$ be the query and $V_1(y) : -B(y)$, $V_2(z, w) : -C(z, w)$ be the view. Assume that Δ contains rules 1) $S(u) : -C(u, e), D(u)$ and 2) $D(m) : -B(m)$. According to the Δ -graph depicted in figure 6.3, providing the path from B to S for algorithm 1, the c-rule $S(u) : -C(u, e), B(u)$

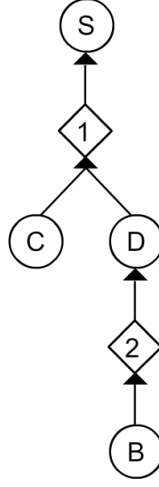


Figure 6.3: A simple Δ -graph

can be constructed. Intuitively, if we show that $C(u, e), B(u)$ is Δ -contained in $S(u)$, then we can use views V_1 and V_2 to rewrite the query. Using Theorem 6.2.3, we can guarantee that the body of a C-rule is always Δ -contained in the head.

Theorem 6.2.3 *Let Δ be a set of ontology rules, and \mathcal{R} be a compound rule, which is constructed w.r.t. a path $\langle r_1 : s_1, r_2 : s_2, \dots, r_n : s_n \rangle$ in Δ -graph. Let Q_1 and Q_2 be two conjunctive queries which are defined as $Q_1(\bar{x}) :- \text{Head}(\mathcal{R})$ and $Q_2(\bar{x}) :- \text{Body}(\mathcal{R})$ where \bar{x} is the same tuple as the head arguments of \mathcal{R} . Then Q_2 is Δ -contained in Q_1 .*

Proof. *Proof is done by induction. Assume the induction base as follows:*

Base: Path $\mathcal{P} = \langle i_1 : k_1, i_2 : k_2 \rangle$ with length $l=2$. Based on the algorithm described in Definition 6.2.5, a compound rule $\mathcal{R}: \text{Head}(\mathcal{R}) :- \text{Body}(\mathcal{R})$ is created when there is a mapping Γ and a homomorphism h such that

$$- \Gamma_1(g_{k_2}) = h(\text{Head}(R_{i_1})) \quad (\text{fact 1})$$

-

$$f(x) = \begin{cases} \Gamma_1(x) & \text{if } x \in \text{Dom}(\Gamma_1) \\ \text{FreshCopy}(x) & \text{Otherwise} \end{cases}$$

- $\text{Body}(\mathcal{R}) = f(\text{Body}(\mathcal{R}_{i_2})/g_{k_2}) \cup \{h(\text{Body}(\mathcal{R}_{i_1}))\}$
- $\text{Head}(\mathcal{R}) = f(\text{Head}(\mathcal{R}_{i_2}))$

To prove the above statement, we need to show that for any variable assignment Z in any interpretation I ($I \models \Delta$), if $\text{Body}(\mathcal{R})^{I,Z} = \text{true}$, then $\text{Head}(\mathcal{R})^{I,Z} = \text{true}$.

Assume an interpretation I_1 where $I_1 \models \Delta$. Since I_1 is an arbitrary interpretation, it can be replaced by any I' consistent with Δ ($I' \models \Delta$). Without losse of generality, assume that R_{i_1} and R_{i_2} are as follows:

$$\begin{aligned} R_{i_1}) \quad P_{1,0}(\bar{y}) &:- P_{1,1}(\bar{x}_1), \dots, P_{1,k_1}(\bar{x}_{k_1}), \dots, P_{1,n}(\bar{x}_n) \\ R_{i_2}) \quad P_{2,0}(\bar{u}) &:- P_{2,1}(\bar{w}_1), \dots, P_{2,k_2}(\bar{w}_{k_2}), \dots, P_{2,m}(\bar{w}_m) \end{aligned}$$

Based on the path, $P_{2,k_2}(\bar{w}_{k_2})$ should be replaced by the body of R_{i_1} . Therefore, the body and the head of \mathcal{R} would be

- $\text{Body}(\mathcal{R}) = f(P_{2,1}(\bar{w}_1)), \dots, f(P_{2,k_2-1}(\bar{w}_{k_2-1})), h(P_{1,1}(\bar{x}_1)), \dots, h(P_{1,n}(\bar{x}_n)), \dots, f(P_{2,m}(\bar{w}_m))$
- $\text{Head}(\mathcal{R}) = f(P_{2,0}(\bar{u}))$

Now, let Z_1 be an arbitrary variable assignment in I_1 such that $\text{Body}(\mathcal{R})^{I_1,Z_1} = \text{true}$.

This means,

$$[f(P_{2,i}(\bar{w}_i))]^{I_1,Z_1} = \text{true} \text{ for } i = 1, \dots, n, \text{ and } i \neq k_2 \quad (\text{fact 2})$$

$$[h(P_{1,j}(\bar{x}_j))]^{I_1,Z_1} = \text{true} \text{ for } j = 1, \dots, n, \quad (\text{fact 3})$$

If we show that $[f(P_{2,k_2}(\bar{w}_{k_2}))]^{I_1,Z_1} = \text{true}$, since $I_1 \models \Delta$, we can immediately infer $[f(P_{2,0}(\bar{u}))]^{I_1,Z_1} = \text{true}$. Back to the definition of f , $f(P_{2,k_2}(\bar{w}_{k_2}))$ is equal to

$\Gamma(P_{2,k_2}(\bar{w}_{k_2}))$. Also, by using the fact 3 and $I_1 \models \Delta$, $[h(P_{1,0}(\bar{y}))]^{I_1, Z_1} = \text{true}$ can be derived. Now, using the fact 1, $\Gamma(P_{2,k_2}(\bar{w}_{k_2}))$ is forced to be true, which leads us to $f(P_{2,k_2}(\bar{w}_{k_2})) = \text{true}$ (according to the definition of function f). (fact 4)

Consequently, by using facts 2, 4, and $I_1 \models \Delta$, we can conclude $[f(P_{2,0}(\bar{u}))]^{I_1, Z_1} = \text{true}$ which is the head of \mathcal{R} . Since I_1 is an arbitrary interpretation, we can derive that for any interpretation I where $I \models \Delta$ and variable assignment Z in I , if $\text{Body}(\mathcal{R})^{I, Z} = \text{true}$, then $\text{Body}(\mathcal{R})^{I, Z} = \text{true}$.

Now, assume the theorem holds for $l=J$. We should show that it also holds for $l=J+1$. This case is straight forward. It can be reduced to the base case by assuming that the Compound rule for $l=J$ is created, and this new rule is added to Δ with id r' . We can change the path from $\langle r_1 : s_1, r_2 : s_2, \dots, r_{J+1} : s_{J+1} \rangle$ to a path $\langle r' : s', r_{J+1} : s_{J+1} \rangle$, and it is already shown that for a path with length two the theorem holds. \square

Definitions 6.2.6 (Complex Compound Rule (CC-rule)) *Complex Compound rule (CC-rule) is a datalog rule which is created by using a D-RAD (i.e., a set of paths) such that the source nodes (i.e. nodes that have no incoming edge) of a D-RAD construct the body, and the head predicate is node \top_D .*

The procedure, represented in the next page, can be used to create cc-rules. The input of this procedure is a set of paths which creates a D-RAD $\{\mathcal{P}\}$, and the output will be a cc-rule in the case of success. It returns nothing in the case of failure.

Algorithm 2 Procedure of creation of cc-rule

```
1: Input: a set of paths  $\{\mathcal{P}\}$  and a set of ontology rules  $\Delta$ 

2: Output: either a c-rule or a cc-rule if it succeeds; nothing if it fails.

3: Create_CC-rule( $\{\mathcal{P}\}, \Delta$ )

4: if ( $|\{\mathcal{P}\}| == 1$ ) then

5:   return C-rule( $\mathcal{P}, \Delta$ )

6: end if

7:  $\mathcal{G} = \emptyset$ 

8: create a new rule  $\mathcal{R}$ 

9: Find  $\mathcal{P}_{T_D} \in \{\mathcal{P}\}$  and remove it from  $\{\mathcal{P}\}$ 

10: for each  $P \in \{\mathcal{P}\}$  do

11:   Find the overlap parts of  $P$  with  $\mathcal{P}_{T_D}$  (e.g.,  $\langle e_k, \dots, e_{k+n} \rangle$ ), and create new path
       $P'$  which contains the non-overlap parts of  $P$ 

12:   if ( $G_{e_k} \in \mathcal{G}$ ) then

13:     put  $P'$  in  $G_{e_k}$ 

14:   else

15:     create a new set  $G_{e_k}$ , and put  $P'$  in it.  $\triangleright G_{e_k}$  is a set labeled with the first
      element in the overlap part

16:     add  $G_{e_k}$  to  $\mathcal{G}$ 

17:   end if

18: end for
```

```

19: for each group  $G_i \in \{\mathcal{G}\}$  do
20:   rule  $R_{G_i} = \text{Create\_CC-rule}(G_i, \Delta)$  ▷ These rules will be used during the
      phase 2 of this algorithm.
21:   if failed to create  $R_{G_i}$  then
22:     terminate and return nothing
23:   end if
24: end for

25: //try to create a rule for  $\mathcal{P}_{\text{TD}}: \langle r_1 : s_1, \dots, r_n : s_n \rangle$  ▷ Phase 2
26: Find  $h, \Gamma$  such that  $\Gamma(g_{s_2}) = h(\text{Head}(R_{r_1}))$ 
27: if ( $h$  and  $\Gamma$  exist) then
28:   create function  $f$  as follows:
      
$$f(x) = \begin{cases} \Gamma(x) & \text{if } x \in \Gamma(x) \\ \text{FreshCopy}(x) & \text{Otherwise} \end{cases}$$


29:    $\text{Body}(\mathcal{R}) = \{f(\text{Subgoal\_Exp}(\mathcal{G}, R_{r_2})/g_{s_2})\} \cup h(\text{Subgoal\_Exp}(\mathcal{G}, R_{r_1}))$ 
30:    $\text{Head}(\mathcal{R}) = \{f(\text{Head}(R_{r_2}))\}$ 
31:   for  $3 \leq i \leq n$  do
32:     Find  $h_i$  and  $\Gamma_i$  such that  $\Gamma_i(g_{s_i}) = h_i(\text{Head}(\mathcal{R}))$ 
33:     if  $h_i$  and  $\Gamma_i$  exist: then
34:       create function  $f_i$  as follows:
          
$$f_i(x) = \begin{cases} \Gamma_i(x) & \text{if } x \in \text{Dom}(\Gamma_i) \\ \text{FreshCopy}_i(x) & \text{Otherwise} \end{cases}$$


35:        $\text{Body}(\mathcal{R}) = h_i(\text{Body}(\mathcal{R}))$ 

```

```

36:         Body( $\mathcal{R}$ ) +=  $\{f(Subgoal\_Exp(\mathcal{G}, R_{r_i})/g_{s_i})\}$ 
37:         Head( $\mathcal{R}$ ) =  $\{f_i(Head(R_{r_i}))\}$ 
38:     else
39:         terminate and return nothing
40:     end if
41: end for
42: return the created cc-rule  $\mathcal{R}$  as the result
43: else
44:     terminate and return nothing.
45: end if

```

▷ End of the algorithm

```

1: Function Subgoal_Exp( $\mathcal{G}, R_{r_i}$ )
2: Body =  $\emptyset$ 
3: for each subgoal  $s_k$  of  $R_{r_i}$  do
4:     if there exists a group  $G_\alpha \in \mathcal{G}$  where  $\alpha$  is equal to  $r_i : s_k$  then
5:         Find  $\delta$  such that  $\delta(Head(R_{G_\alpha})) = s_k$ 
6:         Body +=  $\delta(Body(R_{G_\alpha}))$ 
7:     else
8:         Body +=  $s_k$ 
9:     end if
10: end for
11: return Body

```

6.3 Computing Δ -contained Queries

As discussed in Section 4.4.5, the MiniCon algorithm is our choice to be adopted for Web service composition. The MiniCon produces a set of contained non-recursive rewritten queries such that the union of these queries is maximally contained in the original query. This is in contrast with the inversed-rule algorithm that produces a recursive datalog program. Recursively connecting to some Web services to provide data is not desirable for service consumers, as each connection to Web services may cost them, nor for service providers who are responsible for maintaining the required infrastructures for accessibility of their Web services. In addition, the performance of the MiniCon is higher compared with logic-based algorithms, and it is more suitable to be extended with full dependencies compared with the MCDSAT algorithm. Consequently, the MiniCon is more practicable in comparison with its alternatives. According to Definition 6.2.1, Δ -contained queries are contained in the original query only in the presence of Δ , i.e., we may have more rewritten queries that are contained in the original query. Unfortunately, MiniCon is unable to find these queries due to a lack of support for utilizing the Δ set. Therefore, we extended this algorithm to compute Δ -contained queries as well. In Section 6.3.1, a naive algorithm for this purpose is introduced, and then, our efforts to optimize this algorithm are explained.

6.3.1 MiniCon-FD: A Naive Query Rewriting Algorithm in the Presence of Full Dependencies

6.3.1.1 Motivation

As described in Section 4.4.2.2, during the creation of MCDs, MiniCon considers all the subgoals of all views for each subgoal of the query to check whether they can cover a query subgoal. If there is a mapping from the query subgoal, e.g. g , to the considering view's subgoal, e.g., s in view V , such that by this mapping (i) g and s become identical and (ii) property 1 is satisfied, then a new MCD C is created. This MCD implies that subgoal s in view V can be used to cover the query subgoal g . By the definition of mapping, it can be concluded that if two subgoals g and s have different predicate symbols, then no mapping can be found, causing no covering.

In the presence of the ontology rules (i.e., full dependencies), we may simply express that a concept in the ontology is contained in another concept. In this case, MiniCon fails to use this relation. This kind of failures is explained by Example 6.3.1.

Example 6.3.1 Let $O = \langle L, \Delta \rangle$ be an ontology where $L = \{ \emptyset, \{v_1, \dots, v_{100}\}, \{ Couple, Married, Parent, Mother, hasChild \} \}$, and $\Delta = \{ \langle R_1; Parent(v_1):- Mother(v_1) \rangle, \langle R_2; Parent(v_2):- Couple(v_2), HasChild(v_2, v_3) \rangle, \langle R_3; Couple(v_4):- Married(v_4) \rangle \}$. Let Q and V_1 be as follows:

- $Q(v_5):- Parent(v_5)$
- $V_1(v_6):- Mother(v_6)$

In this domain, all mothers are defined to be parents, like the real world. Since predicate *Parent* and *Mother* are different, no MCD can be created by MiniCon,

resulting in no rewritten query. \square

Example 6.3.1 is a simple instance of the query rewriting in the presence of full dependencies. Because of the expressivity of full dependencies, rewriting queries, e.g. Q in the example 6.3.1, can be more complicated if there exist views such as

- $V_2(v_7) \text{:- } \textit{Married}(v_7), \textit{HasChild}(v_7, v_8)$
- $V_3(v_9) \text{:- } \textit{Married}(v_9)$
- $V_4(v_{10}, v_{11}) \text{:- } \textit{HasChild}(v_{10}, v_{11})$

The idea underlying MiniCon-FD is that when $\text{Predicate}(g)$ is not the same as $\text{Predicate}(s)$, MiniCon-FD retries by finding a rule R which is consistent with Δ , and $\text{Predicate}(s)$ appears in the body and $\text{Predicate}(g)$ appears in the head. MiniCon-FD then attempts to find a mapping τ from g to the head of R . If such a mapping exists, then the query Q will be chased back by replacing g with the body of R , while the body's terms are renamed according to τ .

For instance, in Example 6.3.1, Q can be chased back to form $Q_1^{[1]}(v_1) \text{:- } \textit{Parent}(v_1)$ by using rule 1. Now, view V_1 can be used to rewrite $Q_1^{[1]}$. According to Theorem 6.2.2, we know that $Q_1^{[1]} \sqsubseteq_{\Delta} Q$; hence, $Q'_1(v_1) \text{:- } V_1(v_1)$ would be a contained rewriting of Q . However, Q'_1 is not the only possible rewriting. Q can also be chased back by rule 2, and the result then can be chased backed by rule 3 to form $Q_2^{[2]}(v_1) \text{:- } \textit{Married}(v_1), \textit{HasChild}(v_1, v_{12})$. Now, the contained rewritten queries $Q'_2(v_1) \text{:- } V_2(v_1)$, and $Q'_3(v_1) \text{:- } V_3(v_1), V_4(v_1, v_{13})$ can be produced by MiniCon.

6.3.1.2 The mechanism of MiniCon-FD

To find all the possible chased-back queries, MiniCon-FD finds all the possible D-RADs in the Δ -graph. Then, for each D-RAD, the corresponding CC-rule may be

constructed. Then, a homomorphism function may be defined to rename the terms in the head atom to make this subgoal identical to g . If such a CC-rule and homomorphism exist, the query will be chased back, and the MiniCon algorithm is called to rewrite the chased-back query.

For each CC-rule, only one subgoal of a query can be chased back since the head contains only one subgoal. However, it is possible that more than one subgoal is chased back simultaneously. To handle such cases, a specific basket for each subgoal g in query is created which contains all the possible CC-rules with head g . For chasing back the query, it is enough to choose a rule for each basket. Note that each basket has a default rule “ $g : -g$ ” in order to handle the situations that only some (but not all) subgoals are chased back. The description of the algorithm is provided in Algorithm 3.

Algorithm 3 MiniCon-FD algorithm

```
1: Input: A set of rules  $\Delta$ ,  $\Delta$ -graph, a set of views  $\mathcal{V}$ , query  $Q$ 

2: Output:  $\Delta$ -contained and contained rewritten queries

3: Results= $\emptyset$ 

4: for each subgoal  $g$  in  $Q$  do

5:   create a specific basket  $B_g$ , and add a default rule  $g : -g$  to  $B_g$ 

6:   Find all the possible D-RADs  $\mathcal{D}$  such that  $Predicate(g) = \top_D$ 

7:   for each  $D \in \mathcal{D}$  do

8:     create cc-rule  $\mathcal{R}$ , and find homomorphism  $h$  such that  $h(Head(\mathcal{R})) = g$ 

9:     if  $\mathcal{R}$  and  $h$  exist then put  $h(\mathcal{R})$  in  $B_g$ 

10:    end if

11:  end for

12: end for

13: /*Now we have  $\mathcal{B} = \{B_{g_1}, \dots, B_{g_n}\}$  where  $B_{g_i}$  contains all the consistent rules with
     $\Delta$  where the heads are the same as  $g_i$  in  $Q^*$ */

14:  $Q' = \emptyset$ 

15: for any non-redundant combination of  $\langle R_{g_1}, \dots, R_{g_n} \rangle$  where  $R_{g_i} \in B_{g_i}$  do

16:   add the  $Body(R_{g_i})$  to  $Q'$ , and rename the existential variables in  $Body(R_{g_i})$ 
    with new names which do not appear in  $Q'$ 

17:   Results += MiniCon( $Q'$ ,  $\mathcal{V}$ )

18: end for
```

6.3.1.3 The correctness of MiniCon-FD

We claim that MiniCon-FD constructs rewritten queries which are Δ -contained in a given query. To show that this algorithm is sound, we prove that this claim holds for any output of this algorithm.

To prove the soundness of the algorithm, we can classify the outputs in two disjoint groups based on the selected rules of the basket. The first group includes the produced rewritings which are constructed by using only the default rules of baskets. The second group includes the remaining. For the first group, when the default rule of each basket is selected for chasing back, the chased query is equal to the original query. Thus, for this group, it is sufficient to use MiniCon soundness theorem [74] to show that the provided rewritten queries are contained in the given query. It can be immediately derived that these rewritings are Δ -contained in the given query as well.

For the second group, we know that each query Q' which is constructed by chasing back a given query Q is Δ -contained in Q ($Q' \sqsubseteq_{\Delta} Q$). According to the soundness theorem of MiniCon [74], giving the query Q' as an input to MiniCon, we know that each produced rewritten query Q'' is contained in Q' ($Q'' \sqsubseteq Q'$). It is straightforward that if $Q'' \sqsubseteq Q'$ then $Q'' \sqsubseteq_{\Delta} Q'$. Therefore, $Q'' \sqsubseteq_{\Delta} Q$, and Q'' can be generalized to any rewritten query in the second group.

6.3.2 Optimization of MiniCon-FD

6.3.2.1 MiniCon-FD⁺

The idea behind MiniCon-FD⁺ is to reduce the number of times that MiniCon should be called by reducing the number of rules in each basket. This can be achieved if we

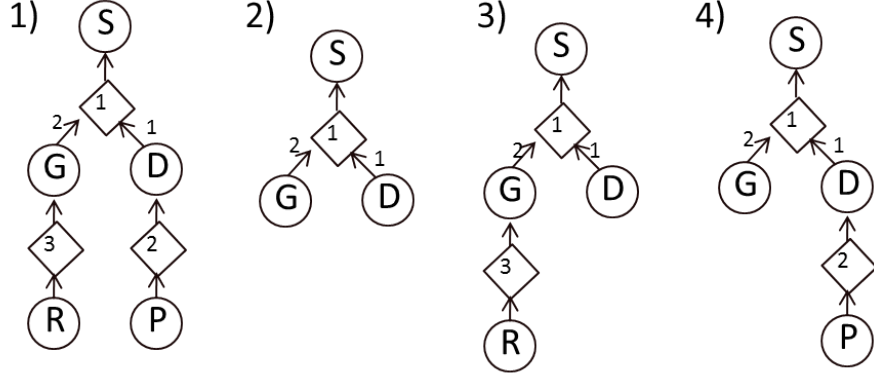


Figure 6.4: Possible D-RADs in the Δ -graph

reduce the number of D-RADs that are required to be considered. Because MiniConFD constructs CC-rule independently without considering the existing views, some CC-rules may be created while there is no view to cover the rules' subgoals. This situation is exemplified by example 6.3.2.

Example 6.3.2 Let $Q(x) : \neg S(x)$ be the query and $V_1(x) : \neg P(x)$ and $V_2(x) : \neg R(x)$ be the views. Let Δ contain the following rules 1) $D(x), G(x) \rightarrow S(x)$, 2) $P(x) \rightarrow D(x)$, and 3) $R(x) \rightarrow G(x)$. Although all the D-RADs drawn in figure 6.4 will be examined, only D-RAD 1 may result in a rewriting.

Examining D-RADs 2, 3, and 4 can be eliminated if we consider only the D-RADs whose source nodes appear in some views. \square

In MiniConFD⁺, only those D-RADs are considered, whose source nodes appear in some views as well. If we have a CC-rule in which a subgoal does not appear in any view, then it is impossible to chase-back the query by the rule and find a contained rewriting due to lack of views which can cover this subgoal.

MiniCon-FD⁺, like MiniCon, explores all the subgoals of the views for each query subgoal. If there is path from a view's subgoal to a query's subgoal, then this path is saved for the next step. In the next step, any possible D-RADs that can be constructed based on the found paths is built. After this part, the procedure of rewriting is the same as MiniCon-FD. Thus, soundness can be proved using the same procedure as the MiniCon-FD soundness proof.

6.3.2.2 MiniCon-FD⁺⁺

In MiniCon-FD⁺, all the subgoals of views are examined for each query's subgoal to check whether there exists a path between them. After finding all the paths, D-RADs and their CC-rules are created based on the paths. MiniCon is then called to rewrite queries created by chasing back the query using these rules, and it again examines all the subgoals of views blindly (duplicate checking). We already collected the information regarding which view can potentially cover which query's subgoal. Thus, if we can use this information, which is gained during the path finding, then we can reduce the consumed time. However, we need to break down the MiniCon algorithm and modify some parts of it, resulting in a more complicated algorithm.

MiniCon-FD⁺⁺, like its ancestor, consists of two main phases: creating the MCDs and combining the MCDs. In contrast to MiniCon, a special class of MCDs are also created which are called partial MCDs, shown by P-MCD. These MCDs are created based on the paths from view subgoals to query subgoals. A P-MCD includes the information about the subgoals of the view that can cover the query subgoal(s), and it consists of two sets: the requirement set and the provide set. The provide set contains the subgoals of the query that can be covered, and the requirement set describes the

needs of this covering that should be provided.

When the algorithm fails to find a mapping from a query subgoal g to a subgoal s of a view V (i.e., failing to use s to cover g), it retries by finding paths from s to g . For each path, the corresponding C-rule is constructed, and then g is expanded. MiniCon-FD⁺⁺ then attempts to find a mapping from a subgoal in the expansion to the view subgoal s . If such a mapping exists, a P-MCD is created such that the provide set contains the subgoal g and all other subgoals in the expansion that are mapped, and the requirement set is filled by the rest of the expansion. P-MCDs are formally defined as follows:

Definitions 6.3.1 (Partial MCD) *For a given subgoal g in the query and subgoal s in view V , a partial MCD P is a tuple $\langle V, \mathfrak{P}, R, Prev, Req, G, h, \varphi \rangle$ where*

- V is view V ,
- \mathfrak{P} is a path from s to g ,
- R is a C-rule created based on path \mathfrak{P} ,
- h is a head homomorphism on view V ,
- φ is a variable mapping from s' in expanded query by rule R to subgoal v such that $\varphi(s') = h(v)$,
- $Prev$ contains the subgoal(s) in the body of R that are covered by mapping φ ,
- Req is a set containing the $Body(R) - Prev$, and
- G is the query subgoal(s) that is covered by mapping φ .

□

The mapping φ of any P-MCD should satisfy Property 1' that is presented in Definition 6.3.2. Moreover, sets $Prev$ and G , similar the procedure of creating MCDs in

MiniCon described in Section 4.4.2.2, are assumed to be minimal.

Definitions 6.3.2 (Property 1') *A mapping φ satisfies property 1', if*

- C1. for each distinguished variable x in the query, $\varphi(x)$ is a distinguished variable.*
- C2. For any non-distinguished variable y in Q , if $\varphi(y) = z$ where z is a non-distinguished variable, then φ should be extended such that any subgoal g in Q that includes y is covered either directly by some subgoals in the view, or by expanding g with a C-rule and then covering all the subgoals g' in the expanded parts that contain y .*

□

In clause C2, when a non-distinguished variable is mapped to a non-distinguished variable in the query ($\varphi(y) = z$), a set $Joint_y$ is created and all the subgoals g in Q that contain y are stored in this set. In the next step, the mapping φ is tried to be extended such that all the subgoals in $Joint_y$ can be covered by some subgoals in the view. If such an extension is possible, an MCD C is created, and $Joint_y$ is added to set G_C . In the case that no subgoal in V covers subgoal g , paths from view's subgoals to g are considered. If such a path exists, g then is expanded by the corresponding c-rule. All the subgoals in the expanded part which contains y are replaced with g in $Joint_y$, and the remains are added to the provide set. If the $Joint_y$ becomes empty, then property 1' is satisfied and a P-MCD will be created; otherwise, we say property 1' is violated. MiniCon-FD⁺⁺ consists of two main phases, creating P-MCD/MCD and combining P-MCD/MCD. First, the mechanism of creation of MCDs (partial and complete) is represented in algorithm 4.

Algorithm 4 MiniCon-FD⁺⁺: P-MCD/MCD creation phase

```
1: Input: A set of rules  $\Delta$ ,  $\Delta$ -graph, a set of views  $\mathcal{V}$ , query  $Q$ 

2: Output: A set of P-MCDs  $\mathcal{P}$  and a set of MCDs  $\mathcal{C}$ 

3:  $\mathcal{P} = \emptyset, \mathcal{C} = \emptyset$ 

4: for each subgoal  $g$  in  $Q$  do

5:   for each subgoal  $s$  in  $V$  do

6:     if  $h$  and  $\varphi$  satisfying property 1' exist and  $\varphi(g) = h(s)$  then

7:       create P-MCD  $P$  based on  $\varphi$ ,  $h$ ,  $Prv$ , and  $Req$  that are populated

       during examining property 1'.

8:     else

9:       if Some path exists from  $s$  to  $g$  then

10:        for each path  $\mathfrak{P}$  do

11:          create c-rule  $R$  and expand  $Q$  by it.

12:          find mapping  $\varphi$  that satisfies property 1', and homomorphism  $h$ 

          such that  $\varphi(g) = h(s)$ .

13:          if  $\varphi$  exists then

14:            create new P-MCD  $P$  and

15:            add to set  $Req_P$  all the subgoals that are produced by the

            expansion, and they are not covered by mapping  $\varphi$ .

16:            add all the covered subgoals to  $Prv_P$  set.
```

```

17:                end if
18:            end for
19:        end if
20:    end if
21:    if P is created and  $Req_P$  is empty then
22:        store P in as a MCD in  $\mathcal{C}$ 
23:    else
24:        store P in  $\mathcal{P}$ 
25:    end if
26: end for
27: end for

```

Example 6.3.3 *Let's assume the ontology $\langle L, \{1\} \text{ Happy}(e):- \text{Smart}(t), \text{SupervisorOf}(t, e) \rangle$, the query $Q(x):- \text{Happy}(x)$, and the views $V_1(z):- \text{Smart}(z)$ and $V_2(u):- \text{SupervisorOf}(u, w)$ all over the same language L defined in Example 4.2.1. The first phase of MiniCon-FD^{++} starts with the first subgoal of the query, $\text{Happy}(x)$ and view V_1 . Due to the lack of mappings from $\text{Happy}(x)$ to $\text{Smart}(z)$, the algorithm finds the paths from Smart to Happy in the Δ -graph and constructs the associated C-rules. Since one path exists, $\text{Happy}(x)$ can be expanded with the body of associated C-rule $\text{Happy}(x):- \text{Smart}(m), \text{SupervisorOf}(m, x)$. Now, a mapping which satisfies property 1' and a head homomorphism must be found such that by using them $\text{Smart}(m)$ and $\text{Smart}(z)$ become identical. The mapping $\varphi = \{m \rightarrow z\}$ and homomorphism $h = \emptyset$ would be sufficient to have $\varphi(\text{Smart}(m)) = h(\text{Smart}(z))$. Thus, a new P-MCD*

P_1 would be created. By following the same steps for view V_2 , the P-MCD P_2 is also created.

$$P_1 = \langle V_1, \langle 1 : 1 \rangle, \text{Happy}(x) \text{:- Smart}(m), \text{SupervisorOf}(m, x), \emptyset, \{m \rightarrow z\}, \{\text{Smart}(m)\}, \{\text{SupervisorOf}(m, x)\}, \{\text{Happy}(x)\} \rangle.$$

$$P_2 = \langle V_2, \langle 2 : 1 \rangle, \text{Happy}(x) \text{:- Smart}(t), \text{SupervisorOf}(t, x), \emptyset, \{t \rightarrow u, x \rightarrow w\}, \{\text{SupervisorOf}(t, x)\}, \{\text{Smart}(t)\}, \{\text{Happy}(x)\} \rangle.$$

□

After the creation of partial and complete MCDs, partial MCDs are first considered. If such a set of P-MCDs exist, such that their *Prev* sets cover all the subgoals in the union of their Req, then a new MCD is created and added to the \mathcal{C} . Note that in these new MCDs, the used view can be the combination of some of the given views. In this case, we can look at these views as a new single view such that the body is conjuncts of these views' bodies, and the head is the union of views' head variables. Therefore, we can use MCD, described in Section 4.4.2.2, without changing its definition.

Considering all the possible combinations is exponential (2^n where n is the number of P-MCDs), making the problem impracticable. Many of these combinations may not result a new MCD; therefore, the P-MCDs paths are again investigated in order to reduce the size of checking-space. The combination of P-MCDs are only considered if their paths create a D-RAD and their C-rules include the same head. To explain the intuition behind these conditions we recall the idea underlying MiniCon-FD⁺.

In MiniCon-FD⁺, if there is a path from a view subgoal to a query subgoal, the path will be saved. After finding all possible paths by investigating all view subgoals, D-RADs based on the found paths are created. For each D-RAD that can be constructed,

the cc-rule is created and by which the corresponding query subgoal is expanded to form a new query Q' , which is Δ -contained in the initial query. Finally, MiniCon is called to rewrite Q' . The idea in MiniCon-FD⁺⁺ is also the same, but with different operation. In MiniCon-FD⁺⁺, our goal is also to expand the query with only those c-rules that all the source nodes (i.e. nodes that have no incoming edge) of the associated D-RADs are covered by some views' subgoals. Therefore, we investigate only those subset of P-MCDs that their paths create a D-RAD with the same c-rule's head to guarantee all the paths belong to the same derivation tree. If such a set of P-MCDs exist, the algorithm tries to find a homomorphism h and a mapping ϕ from the Prv sets to Req sets while ϕ is consistent with φ 's, that means the queries variables must be mapped to the same query variables. This part distinguishes MiniCon-FD⁺⁺ from MiniCon-FD⁺, when we use the information collected during investigation of paths instead of re-investigating the views subgoals.

When all the subset of P-MCDs which satisfy the above conditions are investigated, and the new MCDs are created, the algorithm performs the same procedure as MiniCon (see Section 4.4.2.2), to combine the MCDs. Consequently, the only part which needs to be explained is the procedure of combining the P-MCDs which is represented in algorithm 5.

Algorithm 5 MiniCon-FD⁺⁺: Combining P-MCDs

```
1: Input: Two P-MCDs  $P_1$  and  $P_2$ ;

2: Output: a set of new P-MCDs  $\mathcal{P}$  and MCDs  $\mathcal{C}$ ;

3:  $\mathcal{C} = \emptyset, \mathcal{P} = \emptyset$ ;

4: for each subgoal  $g$  in  $Req_{P_2}$  do

5:   for each subgoal  $s$  in  $Prv_{P_1}$  do

6:     Find a variable mapping  $\phi$  from  $Prv_{P_1}$  to  $Req_{P_2}$  such that

7:     (E1)  $\phi(x) = x$  where  $x$  is a query variable and (E2)  $\phi(s) = g$ ;

8:     if  $\phi$  exists then

9:        $Req_{P_2} = Req_{P_2} - \{g\}$ ;

10:       $Req_{P_1} = \phi(Req_{P_1}) - Prv_{P_2}$ ;  $\triangleright$  Rename variables in  $Req_{P_1}$  w.r.t  $\phi$ 

11:      create new P-MCD  $P_3$  such that

12:       $V_{P_3} = V_{P_1} \cup V_{P_2}$ ;

13:       $Prv_{P_3} = \phi(Prv_{P_1}) \cup Prv_{P_2}$ , and  $Req_{P_3} = Req_{P_1} \cup Req_{P_2}$ ;

14:       $h_{P_3} = \phi(h_{P_1}) \cup h_{P_2}$ , and  $\varphi_{P_3} = \phi(\varphi_{P_1}) \cup \varphi_{P_2}$ ;

15:       $Path_{P_3} = Path_{P_1} \cup Path_{P_2}$ ;

16:       $R_{P_3} = Create\_CC - rule(Path_{P_3})$ ;  $\triangleright$  (see algorithm 3)

17:      if  $R_{P_3}$  exists and  $Req_{P_3} = \emptyset$  then create new MCD  $C$  w.r.t  $P_3$ , and

        add it to  $\mathcal{C}$ ;

18:      end if

19:      if  $R_{P_3}$  exists and  $Req_{P_3} \neq \emptyset$  then add  $P_3$  to  $\mathcal{P}$ ;

20:      end if

21:    end if

22:  end for

23: end for
```

Example 6.3.4 To combine P-MCDs P_1 and P_2 constructed in example 6.3.3, a mapping ϕ is defined from Prv_{P_1} , $\{Smart(m)\}$, to Req_{P_2} , i.e. $Smart(t)$. Since m and t are not query variables, we would have $\phi = \{m \rightarrow t\}$ which satisfies conditions E1 and E2. Therefore, $Smart(t)$ should be removed from Req_{P_2} , and $Req_{P_1} = \{\phi(SupervisorOf(m, x))\}$ - $SupervisorOf(t, x) = \emptyset$, according to lines 9 and 10. A new P-MCD P_3 is created as follows:

- $Req_{P_3} = Req_{P_1} \cup Req_{P_2} = \emptyset$
- $Prv_{P_3} = \phi(Prv_{P_1}) \cup Prv_{P_2} = \{Smart(t), SupervisorOf(t, x)\}$
- $h_{P_3} = \emptyset$
- $\varphi_{P_3} = \phi(\{m \rightarrow z\}) \cup \{t \rightarrow u, x \rightarrow w\} = \{t \rightarrow z, t \rightarrow u, x \rightarrow w\}$
- $V_{P_3} = \{V_1, V_2\}$
- $Path_{P_3} = \{\langle 1 : 1 \rangle, \langle 2 : 1 \rangle\}$
- $R_{P_3} = Happy(x) :- Smart(t), SupervisorOf(t, x)$
- $G_{P_3} = \{Happy(x)\}$

Because R_{P_3} exists and Req_{P_3} is empty, a new MCD $C_1 = \langle h_{P_3}, V_{P_3}, \varphi_{P_3}, G_{P_3} \rangle$ is created and added to MCD set. Finally, the set $\mathcal{C} = \{C_1\}$ is passed to the combination phase of MiniCon, resulting in the rewritten query $Q'(x) : -V_1(x), V_2(x)$.

□

6.4 Conclusion

MiniCon-FD is a naive algorithm that rewrites queries in the presence of full dependencies and produces Δ -contained queries. Given an ontology $\langle \mathcal{L}, \Delta \rangle$, it creates a basket for each query subgoal and fills it with all the CC-rules in which the head is

the same as the query subgoal. Because, the Δ -graph is a finite tree, the number of D-RADs, which are subtrees in the Δ -graph, are finite. Therefore, the size of baskets are bounded, and the calculation will terminate. In the next step, a cc-rule is selected from each basket, and the query is expanded by using it if possible in order to form query Q' . Then, Minicon is called to rewrite Q' . Since Q' is Δ -contained in the initial query, the rewritings are Δ -contained rewritten queries. However, due to independently construction of CC-rules without considering the existing views, the irrelevant cc-rules can be examined which is exemplified by example 6.3.2.

Considering views during constructing D-RADs is the idea underlying MiniCon-FD⁺. In this algorithm, when a D-RAD is found, the algorithm first checks that at-least one view exists for each leaf such that a subgoal predicate of the view is the same as the leaf label. Time for this check is $O(\log(n \times m \times r))$ where n is the number of source nodes in the D-RAD, m is the number of views, and r is the maximum number of subgoals in a view. Compared with the time which is required for executing MiniCon to rewrite, this check can efficiently reduce the time in practice by reducing the number of expanded queries with cc-rules that should be rewritten by MiniCon. Nonetheless, there is no guarantee that a D-RAD that passed the check will lead us to a valid rewritten query.

In MiniCon-FD⁺, during the process of finding paths, pieces of information, for example, the third subgoal of n^{th} view can cover the second query subgoal, are gained. Nonetheless, these pieces of information are re-collected during running the MiniCon due to the distinct execution of these two phases. To avoid the redundant collection of information, MiniCon-FD⁺⁺ is introduced which is integrated with MiniCon and modified some parts of it.

Integrating with MiniCon algorithm improves the performance of MiniCon-FD⁺⁺ in compared with MiniCon-FD⁺. On the other hand, since MiniCon-FD⁺ is distinct from MiniCon, it can be added to other rewriting algorithms such as GQR algorithm (explained in Section 4.4.4) which has a better performance in practice, resulting in a faster query rewriting algorithm in the presence of full dependencies.

In the domains where the ontology is fixed, the performance of MiniCon-FD and its descendants can also be improved by offline calculation of D-RADs and their associated rules. By assuming the ontology and its Δ -graph as fixed, all the possible the D-RADs and their cc-rules can be calculated and stored offline. Hence, in online process when a query is posed, the system only requires to find the related calculated cc-rules which can speed up the rewriting process.

Chapter 7

Preference-based Composition of Web Services

7.1 Adopting MiniCon-FD⁺ for Web Service Composition

Web services can be classified into two major groups of data providing (DP) and state-changing services based on their outputs and effects after execution. DP Web services receive data as their input (possibly empty input) and provide data as the output. For example, a service that provides a list of departure flights from a given airport is categorized as a DP service. State-changing services not only provide data, but may also change the state of the world. The state of the world can generally be defined as a set of parameters and variables that describe the situations such as before and after the execution of a Web service. A flight booking service is categorized as a state-changing service since it can change the number of available seats, resulting in

a change in the state of the system after execution.

This thesis is mainly focused on DP services. Therefore, in our work, Web services can be seen as views, and Web service composition can be seen as the integration of these views. Consequently, MiniCon-FD⁺, introduced in Section 6.3.2.1, can be used for the purpose of Web service composition. However, since there is no method to specify input and output parameters, modifications are required in the integration system. In this Chapter, Web services and queries are first defined, and then MiniCon-FD⁺ is adapted for composing Web services. Finally, a formal framework is designed to encode user preferences and to rank the results of a composition based on these preferences.

7.1.1 Queries and Web services

In the Web service composition problem, a query is received to describe a desired Web service. Since Web services can receive input and output parameters, a query should capture these pieces of information. Outputs are the data resulting from execution of a Web service, and inputs are defined as some pieces of information that are required to generate outputs.

In this approach, queries are described in the form of conjunctive queries (see Section 4.2) over a given ontology's language \mathcal{L} , which is described in Definition 6.1.1. To distinguish the inputs from the outputs, the character \$ is used. The terms in the head that have a \$ sign are the inputs, indicating that they can be provided for the desired Web service. The entire head tuple, consisting of inputs and the other head terms, are defined as the expected outputs of the desired Web service.

Definition 7.1.1 Queries *A query Q over the ontology language \mathcal{L} is a conjunctive query in the form of*

$$Q(\$ \bar{g}, \bar{y}) : -R_1((\bar{x}_1)), \dots, R_k((\bar{x}_k))$$

Where \bar{g} and \bar{y} are the lists containing \mathcal{L} -terms. \bar{g} represents the inputs, and tuple $(\$ \bar{g}, \bar{y})$ represents the outputs of a desired Web service. \square

As mentioned in Section 2.3, each Web service needs a description to represent its functional characteristics. This description can be modeled using views syntax. However, because of input and output parameters, this syntax needs to be modified; thus a Web service description is defined as follows:

Definition 7.1.2 Web services *Given an ontology language \mathcal{L} , a Web service is described similar to the queries as a conjunctive query over \mathcal{L} in the form of*

$$V(\$ \bar{g}, \bar{y}) : -R_1((\bar{x}_1)), \dots, R_k((\bar{x}_k))$$

Where \bar{g} and \bar{y} are lists of \mathcal{L} -terms and represent the inputs and outputs of the Web service, respectively. In contrast to queries, V is assumed to be a unique name used to refer to a real Web service, and it is stored in a set \mathcal{V} called Web service registry.

\square

In addition to functional characteristics, which are captured by conjunctive queries, each Web service requires descriptions of service contract, its location, and how it can be invoked (as mentioned in Section 2.2.3). The structure of these is out of the scope of this research. We assume that this information is provided by the service publisher and stored in a service registry agency, and can later be retrieved by using

the name V . Therefore, each Web service registered in our service registry requires a unique name which is the same as the head of the conjunctive query V .

7.1.2 Web Service Composition

MiniCon-FD⁺ can be used to compose Web services when no input is defined. In the presence of inputs for Web services, some of the results cannot be executed in order to produce the expected outputs. An example is provided below (Example 7.1.1), demonstrating a case in which the required input of a Web service in a composition cannot be provided.

Example 7.1.1 Consider the query Q , views V_1 and V_2 that are registered in a service registry, and the rewritten query Q' as the result of MiniCon-FD⁺ if the input parameter is ignored.

- $Q(\$x, y) : -R(x, y, z), S(y, z)$
- $V_1(\$x, \$y, z) : -R(x, y, z), P(x, y)$
- $V_2(\$y, z) : -S(y, z)$
- $Q'(\$x, y) : -V_1(x, y, z), V_2(y, z)$

In the rewritten query Q , V_1 and V_2 require y as the input to provide output, while y is not the input of the query. It cannot be provided by any other Web services in Q' neither; hence no possible order exists for the execution of V_1 and V_2 to produce the output. Nevertheless, this rewritten query is a contained rewriting in the absence of input output parameters. \square

Handling input and output parameters in the query rewriting process has been studied in the data integration area. This problem is known as query rewriting in the presence

of an access pattern limitations, where there may exist some limitations to access data [76]. For instance, to get the current temperature from Yahoo! Weather, it is not possible to ask for all the tuples in the database. Instead, the received query should specify some parameters, such as the location, to get the temperature values.

In the data integration area, access pattern limitation is modeled by attaching a string to views and queries. For each view or query, a string with length n is attached where n is the number of terms in the head. and k^{th} letter in the string indicates the type of the k^{th} term head. Terms can be either *free* or *bound* which are shown by letters f and b , respectively. Bound terms appearing in a view imply that the view can only provide tuples as the outputs if values for the bound terms are provided. In contrast, free terms make no limitation for providing outputs. To apply this model to Example 7.1.1, view V_1 can be modeled as $V_1^{bbf}(x, y, z)$. In this thesis, we selected the \$-sign model to encode the access patterns.

The complexity of query rewriting in the presence of access pattern limitation has been considered in several works. In [76], it is shown that the bound mentioned in Theorem 4.3.1 does not hold for the length of rewritings in the presence of access pattern limitations (see Example 7.1.2). They also showed that in the case of finding the equivalent rewriting, it is enough to consider the rewritten queries with length of $n + v$ where n is the number of query subgoals and v is the number of variables in the query. In the case of finding maximally-contained rewriting, the authors in [50] showed that no bound exists for the length of rewritings. In [24], the authors proposed an algorithm to find maximally-contained rewritings in the presence of access pattern limitations by producing recursive rewritings.

Example 7.1.2 Consider the following query and views. The query asks for the persons who are supervised by John. V_1 provides all the supervisors of the given person x , and V_2 provides all the persons who are supervised.

- $Q(x) : \neg SupervisorOf(x, 'John')$
- $V_1(\$x, y) : \neg SupervisorOf(x, y)$
- $V_2(x) : \neg SupervisorOf(x, y)$
- $Q(x) : \neg V_2(x), V_1(\$x, 'John')$

The rewritten query Q' can provide the tuples that are requested by the query. However, based on Theorem 4.3.1, the length of a rewritten query, if exists, should be less than the size of the query. \square

In this approach, we focus on non-recursive query rewriting in the presence of access pattern limitation, and we consider the rewritten queries with the length of smaller or equal to the size of the initial query. Our system first rewrites a given query without considering input parameters, and it then verifies whether a produced rewriting is executable by checking the input parameters. If there is an order for the views used in a rewriting such that the inputs of all the views can be provided either by the user or by the earlier views, then this rewriting is added to the verified set. The formal definition of executability of a rewriting is described in Definition 7.1.3.

Definition 7.1.3 (Executable rewriting). *Given a query Q , a rewritten query Q' is executable if there is an acyclic order for the execution the of used views in Q' such that the inputs of each view can be provided either*

- *by the user, i.e., the input parameters in the used view are also labeled as the inputs in Q , or*

- by the earlier views in the order.

□

Intuitively, users expect not to have a composition that always provides an empty set as the output. Although it is impossible to guarantee a composition that always has some outputs, we can avoid providing compositions that always provide an empty set in the presence of full dependencies. For example, assume we are looking for kind and supportive persons (i.e., $Q(x) : \neg \text{Supportive}(x), \text{Kind}(x)$), and let $V_1(x) : \neg \text{Mother}(x)$ and $V_2(x) : \neg \text{Father}(x)$ be the views. By assuming the ontology rules $\Delta = \{1) \text{Mother}(x) \rightarrow \text{Kind}(x), 2) \text{Father}(x) \rightarrow \text{Supportive}(x), 3) \text{Mother}(x), \text{Father}(x) \rightarrow \perp\}$, MiniCon-FD⁺ will provide the following rewritten query $Q'(x) : \neg V_1(x), V_2(x)$. However, it can be derived from the ontology rules that Q' always produces empty set if executed. In this case, we call V_1 and V_2 as impossible views based on the ontology rules.

Definition 7.1.4 (Impossible views). *Given a domain ontology $\langle \mathcal{L}, \Delta \rangle$, two views V_1 and V_2 used in a rewriting Q' are impossible if $\Delta \not\models \top \rightarrow \text{exp}(V_1) \wedge \text{exp}(V_2)$ where $\text{exp}(V)$ is the conjunction of subgoals resulted from replacing the V 's head with its body in $\text{Exp}(Q')$. □*

After ensuring the executability of a rewriting, each two consecutive views should be tested to not be impossible. This test can be done by expanding a rewritten query, and verifying its consistency with the ontologies rules.

The order of performing these two tests depends on their computational complexity because each test is performed after running MiniCon-FD⁺, and it may also reduce

the number of rewritings. The more complex test can be performed after the results set is reduced by the simpler one. The complexity of testing the executability of a rewriting highly depends to the number of subgoals and the number of input variables in the views. The test of impossibility of views, depends on the size of Δ in the domain ontology. Therefore, based on the domain and the views, this order can change.

Up to this point, definitions of queries and views are modified to handle DP Web services. Two more extensions are required to verify the results of rewriting produced by MiniCon-FD⁺ due to the access pattern limitations as well as rules with the form of $P_1(x_1), \dots, P_n(x_n) \rightarrow \perp$. Using Definitions 7.1.3 and 7.1.4, we can formally define composed Web services.

Definition 7.1.5 (Composed Web services). *Given a domain ontology O , a query Q , $\{pref\}$ a set of user preferences (see definition 7.2.4), and a set of Web services all over the ontology language, a composed Web service CWS is a tuple $\langle \{\mathcal{R}\}, \{\mathcal{C}\}, Cmp, TC \rangle$ where*

- $\{\mathcal{R}\}$ is a set of CC-rules used to expand Q in order to form Q' ,
- Cmp is an executable rewriting of Q' such that no pair of views used in Cmp is impossible,
- $\{\mathcal{C}\}$ is a set of MCDs used to construct Cmp , and
- TC is a zero or positive number indicating the total cost of violations by the considering Cmp by the evaluated preferences.

□

The TC is calculated during the evaluation of user preferences to rank the composed Web services. The processes of evaluation and ranking are explained in Section 7.2.3. In the next Section, the user preferences and their syntax and semantics are described. Next, to evaluate our system, the example in Chapter 3 is implemented.

7.2 User Preferences

User preferences are defined as alternative objectives that are not the goals of the composition, but we are interested to satisfy them as much as possible. To evaluate preferences, an expressive and tractable language is required. This language needs to be expressive enough to capture all the information about the users' preferences. The evaluation of statements in this language also needs to be tractable, so that we can rank the results in a reasonable time. In this study, the datalog language is selected to describe the preference formulas because of its tractability . In addition, it can easily be integrated with ontology rules to derive more information without additional translation.

Datalog language, by itself, is not powerful enough to describe some preferences shown in Section 3.3. For instance, the preference for the fourth desired Web service cannot be expressed in datalog language because there is no cardinality restriction defined in this language. Therefore, we design a formal framework to expand the expressivity of datalog language.

In addition, to enhance the capability of the framework to encode the user preferences accurately, users should also be able to specify the fragment of their query which is related to their preferences.. For instance, consider example 7.2.1. In this example,

if we can specify that the preference $CanandianCity('Toronto')$ is related to subgoal $CanadianCity(c2)$ we can then use this preference statement to accurately rank the compositions. Otherwise, compositions that provide tuples of student and professors such that the professors are born in Toronto will satisfy the preference as well.

Example 7.2.1 Consider a query $Q(x,y):- Student(x), Professor(y), BornIn(x,c1), BornIn(x,c2), CanandianCity(c1), CanadianCity(c2)$ which requests for students and professors who were born in Canada. Lets assume our preference is to find students who were born in Toronto. A preference such as $CanandianCity('Toronto')$ cannot properly encode the meaning of this preference because the query subgoal to which $CanandianCity('Toronto')$ refers is not clearly indicated. \square

In this framework, two classes of preferences are described: *global preferences* and *local preferences*. Global preferences are those preferences that are associated with the entire query, while local preferences are associated with a fragment of the query. These two classes bring more expressivity and not only enable the users to describe complex preferences, such as the third preference described in Chapter 3, but they also assist in correctly ranking the results of the composition.

Since users describe the query as well as the preferences, they are able to specify the parts of the query which are associated with their preference. To handle local and global preferences, a proper language is required for describing preferences. The syntax of this language is described in Section 7.2.1.

7.2.1 Syntax of User Preferences

In this Section, the syntax of our language for describing user preferences is defined. The structure of this syntax is explained from basic to complex preference formulas. The core part of the preference language is in the form of datalog rules and is defined as follows:

Definition 7.2.1 (Basic preference formula). *Given an ontology language $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, a basic preference formula f is defined as*

$$P_2(\bar{x}_2), \dots, P_k(\bar{x}_k) \rightarrow P_1(\bar{x}_1) \mid \top \rightarrow P_1(\bar{x}_1) \mid P_2(\bar{x}_2), \dots, P_k(\bar{x}_k) \rightarrow \perp$$

where $P_i(\bar{x}_i)$ is an \mathcal{L} -atoms. \square

According to definition 7.2.1, the preference described in Example 7.2.1 can be $\top \rightarrow \text{CanandianCity}(\text{'Toronto'})$. As another example, if a user prefers to find students who were not born in Ontario, she can use $\text{Ontario}(c1) \rightarrow \perp$.

To distinguish local preferences from global preferences, the associated part of the query to the preference formula should be encoded, which is done by defining preference bodies.

Definition 7.2.2 (Preference body). *Given an ontology language \mathcal{L} and a query Q over \mathcal{L} , a preference body is a tuple $\langle f, sq \rangle$ where f is a basic preference formula over \mathcal{L} , and sq is a subset of subgoals in Q . pb is a unique name to refer to the preference body, and it is stored in a set which is called preference body names and is shown by PB . \square*

Example 7.2.2 Recalling from Example 7.2.1, the preference of finding students who were born in Toronto can be encoded by preference body pb such that $pb = \langle \top \rightarrow CanandianCity('Toronto'), CanandianCity(c1) \rangle$. \square

Local preferences can improve accuracy in evaluating the preferences. Nevertheless, they are not sufficient yet to express more complex preferences such as *either student x is preferred to be born in Alberta or if x was born in Ontario then professor y should be born in Ontario too, for each pair of (x,y) in Example 7.2.1.*

To make the language more expressive, the class of complex preference formulas is defined. Complex formulas are defined over the preference body names. The definition of complex preference formulas is provided in Definition 7.2.3.

Definition 7.2.3 (Complex preference formula). *The language for describing a complex preference formula is a tuple $\langle PB, \{\vee, \wedge, \rightarrow, n \preceq \odot\} \rangle$ where n is a positive number and PB is the set of all preference bodies' names, and the second argument is the operator set. Given such a language, a complex preference formula cf is defined as*

$$pb \mid cf_1 \wedge cf_2 \mid cf_1 \vee cf_2 \mid cf_1 \rightarrow cf_2 \mid n \preceq cf_1 \odot \dots \odot cf_{n+m}$$

where m and n are positive numbers, $pb \in PB$, and cf_i is a complex preference formula. \square

Example 7.2.3 Recalling the query from Example 7.2.1, to express the preference student x is preferred to be born in Alberta, or if x was born in Ontario the professor y should be born in Ontario too, the following preference bodies can be defined:

$$pb_1 = \langle \top \rightarrow Alberta(c1), \{CanadianCity(c1)\} \rangle$$

$$pb_2 = \langle \text{Ontario}(c1) \rightarrow \text{Ontario}(c2), \{\text{CanadianCity}(c1), \text{CanadianCity}(c2)\} \rangle$$

The complex preference formula CF_1 can be defined as $CF_1 = pb_1 \vee pb_2$. \square

To enable users to prioritize their preferences, users associate their preference with an integer value. This value is called the cost of preference and is collected when the preference is violated by a composition. Recalling Definition 7.1.5, a composition is a tuple $\langle \{\mathcal{R}\}, \{\mathcal{C}\}, \text{Cmp}, TC \rangle$ where TC is a positive number indicating the total costs of violations of the evaluated preferences by the composition. During ranking of the composed Web services, if a composition violates a preference, the cost value associated to the preference is added to the composition's TC .

Definition 7.2.4 (Preferences). *Given an ontology language \mathcal{L} and a set of preference bodies in terms of \mathcal{L} which their names are stored in PB , a preference is a tuple $\langle cf, C \rangle$ where cf is a complex formula in terms of PB , and C is a number which indicates the cost that should be collected if cf is violated by a composition. The value cf is determined by the user, and it must be greater than one. \square*

7.2.2 Semantics of User Preferences

The semantics of preferences is model-theoretic and is achieved by assigning a truth value to each preference body and then evaluating the complex preference formulas with respect to these values. As mentioned before, two classes of preferences, i.e., global and local preferences, can be defined. To assign truth values to a preference body $\langle f, sq \rangle$ of a global preference, since sq indicates the entire query, it is enough to check whether the entire rewritten query is consistent with f w.r.t the domain rules Δ . In contrast, assigning truth values to local preference bodies is not simple.

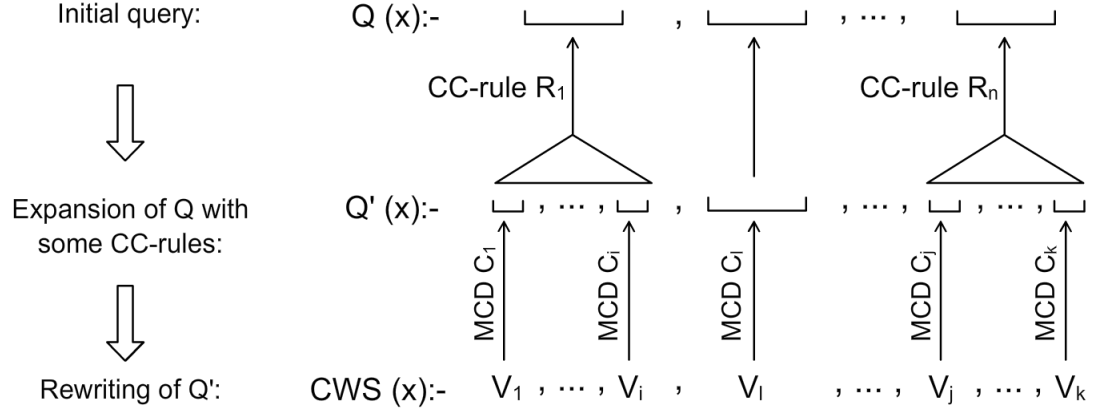


Figure 7.1: Relations between query subgoals and rewritten query subgoals

To assign truth value to a local preference body $\langle f, sq \rangle$, we first need to know the part of the composed Web service that is used to cover q . Sets $\{\mathcal{R}\}$ and $\{\mathcal{C}\}$ of a composed Web service are utilized to determine the parts that cover q . As shown in Figure 7.1, since each MCD contains a mapping from some subgoal s in Q' to some view subgoals, we can use MCDs to determine the rewritten query subgoals that cover s . On the other hand, using the utilized CC-rules to form Q' , we can determine whether a subgoal in Q' is the result of the replacement of a head by the body of CC-rule, and we can also detect the replaced subgoals. Consequently, given a query and a composed Web service, for any query subgoal s , we can determine the set of subgoals in a composition that covers s . We call this set $Coverall_q$. In addition to these subgoals, all the other subgoals in the rewritten query that have shared variables with $Coverall_q$ are also added to this set.

Example 7.2.4 Consider a query $Q(x,y):-$ Student(x), Professor(y), BornIn($x,c1$), BornIn($y,c2$), CanadianCity($c1$), CanadianCity($c2$) which requests for students and

professors who were born in Canada. Let assume our preference is to find students who were born in Toronto. A preference such as *CanandianCity*(‘Toronto’) cannot properly encode the meaning of this preference because the query subgoal to which *CanandianCity*(‘Toronto’) refers is not clearly indicated. \square

Definition 7.2.5 (*Coverall_q*). *Given an ontology $\langle \mathcal{L}, \Delta \rangle$, query Q , and a composition $CWS = \langle \{\mathcal{R}\}, \{\mathcal{C}\}, Cmp, TC \rangle$, all w.r.t \mathcal{L} , for a subgoal q in Q , *Coverall_q* is the set of subgoals in Cmp such that*

- C1 they either cover q directly or the subgoals which are resulted by expanding q with a CC-rule in $\{\mathcal{R}\}$, or*
- C2 they have some shared variable(s) with the subgoals collected in clause C1.*

\square

We later use *Coverall_q* to assign truth values to preference bodies. For a global preference body, *Coverall_q* is the entire rewritten query, while a fragment of a rewritten query may be in *Coverall_q* of a local one.

Definition 7.2.6 (Basic preference formula semantics). *Given an ontology $\langle \mathcal{L}, \Delta \rangle$ and a set of atoms A , for any interpretation $I \models \Delta$, the semantics of a basic preference formula f is defined as*

- $[P_2(\bar{x}_2), \dots, P_k(\bar{x}_k) \rightarrow P_1(\bar{x}_1)]^I = \text{true}$, if either $(P_2(\bar{x}_2), \dots, P_k(\bar{x}_k))^{I,Z} = \text{false}$ or $P_1^{I,Z} = \text{true}$ for all the variable assignments Z in I by which all the atoms $s \in A$ $s^{I,Z} = \text{true}$.
- $[P_2(\bar{x}_2), \dots, P_k(\bar{x}_k) \rightarrow \perp]^I = \text{true}$ if $[P_2(\bar{x}_2), \dots, P_k(\bar{x}_k)]^{I,Z} = \text{false}$ for all the variable assignments Z in I by which all the atoms $s \in A$ $s^{I,Z} = \text{true}$.

- $[\top \rightarrow P_1(\bar{x}_1)]^I = \text{true}$, if $P_1(\bar{x}_1)^{I,Z} = \text{true}$ for all the variable assignments Z in I by which all the atoms $s \in A$ $s^{I,Z} = \text{true}$.

If formula f is evaluated as true (i.e., f is satisfied) under I , it is then shown by $I \models_{\Delta \cup A} f$. \square

By using the semantics of basic preference formula, the semantics of preference bodies and complex preference formulas are defined in Definitions 7.2.7 and 7.2.8, respectively.

Definition 7.2.7 (Preference body semantics). *Given an ontology $\langle \mathcal{L}, \Delta \rangle$ and the union of $Coverall_q$ for each $q \in sq$ which are constructed w.r.t. preference body $pb = \langle f, sq \rangle$ and a composition $CWS = \langle \{\mathcal{R}\}, \{\mathcal{C}\}, Cmp, TC \rangle$, pb is evaluated as true if for any interpretation I , $I \models_{\Delta \cup Coverall_q} f$; otherwise, pb evaluates to false. Moreover, a model M_{CWS} is created for the preference body names such that $M_{CWS} \models pb$ if $pb = \text{true}$. Otherwise, $M_{CWS} \not\models pb$. \square*

The semantics of complex preference formulas are defined with respect to the model M_{CWS} . Therefore, all the preference bodies are first evaluated, and a model M_{CWS} is created based on their truth values. In other word, M_{CWS} is a model for a propositional logic language L where the predicates in this language are the preference bodies' names. After constructing the model M w.r.t a composition CWS , the complex formulas can be evaluated using M_{CWS} because they are defined over the same terms as M_{CWS} , which are preference bodies' names.

Definition 7.2.8 (Complex preference formula semantics). *Given a set of preference body names PB and a model M_{CWS} constructed w.r.t. to PB , A complex preference formula cf is satisfied by the model M_{CWS} ($M_{CWS} \models cf$) if:*

- $M_{CWS} \models cf$ if $M_{CWS} \models pb$ where $cf = pb$, and $pb \in PB$
- $M_{CWS} \models cf_1 \wedge cf_2$ if $M_{CWS} \models cf_1$ and $M_{CWS} \models cf_2$ where cf_1 and cf_2 are complex formulas.
- $M_{CWS} \models cf_1 \vee cf_2$ if either $M_{CWS} \models cf_1$ or $M_{CWS} \models cf_2$ where cf_1 and cf_2 are complex formulas.
- $M_{CWS} \models cf_1 \rightarrow cf_2$ if either $M_{CWS} \not\models cf_1$ or $M_{CWS} \models cf_2$ where cf_1 and cf_2 are complex formulas.
- $M_{CWS} \models n \preceq cf_1 \odot \dots \odot cf_{n+m}$ if $|\{cf_i \mid M_{CWS} \models cf_i, 1 \leq i \leq (m+n)\}| \geq n$ where cf_i is a complex formula.

□

Definition 7.2.9 Preference semantics. A preference $\langle cf, C \rangle$ is satisfied by a composed Web service CWS , if and only if, $M_{CWS} \models cf$. □

7.2.3 Evaluation of User Preferences

Users describe a query as well as their preferences for a desired composition, based on Definitions 7.2.1-7.2.4, and then send them to the framework. The framework evaluates all the preferences for each composition and collects the cost of violated preferences. It finally ranks the compositions based on their associated total cost.

To commence the evaluation, the described preference bodies are first considered. As explained in Definition 7.2.2, a preference body pb consists of a formula f and subset of query subgoals q such that q specifies the part of the query that should be considered for satisfying the formula f . In addition, the relations between the subgoals of the original query and the subgoals in the expanded rewritten query can be determined by

considering the MCDs and CC-rules used to construct a composition. The subgoals in the expanded rewritten query that cover q are collected in a set called $Coverall_q$. Given a preference body $pb=\langle f, sq \rangle$ w.r.t. an ontology domain $\langle \mathcal{L}, \Delta \rangle$, for any interpretation I (i.e., any database) that is consistent with our ontology rules (Δ), if all the subgoals in $Coverall_q$ are evaluated as true, then f is also evaluated as true. In other words, if the evaluation process is translated to first-order logic, the following FO-sentence can simply describe this process:

$$\forall \bar{x} \forall \bar{y} \forall \bar{z} \bigwedge Coverall_q(\bar{x}, \bar{y}) \rightarrow f(\bar{x}, \bar{z})$$

where \bar{x} , \bar{y} , and \bar{z} are \mathcal{L} -variables that appear in f and $Coverall_q$. $\bigwedge Coverall_q(\bar{x}, \bar{y})$ means the conjunction of the subgoals in $Coverall_q$.

For each composed Web service CWS , all preference bodies should be evaluated because the set $Coverall_q$ completely depends to the CWS . During the evaluation of all the preference bodies, a model M_{CWS} is also created according to the truth values of evaluated preference bodies. After this step, only model M_{CWS} is used to evaluate the preference complex formulas, and consequently the preferences are evaluated.

Definition 7.2.10 (Evaluation of preferences). *For a given preference $pref=\langle cf, C \rangle$ and a composition $CWS=\langle \{\mathcal{R}\}, \{\mathcal{C}\}, Cmp, TC \rangle$, the associated cost of the preference C is collected if and only if cf is violated by CWS that means*

$$TC+ = C \text{ if and only if, } CWS \not\models cf.$$

□

Another additional feature is defined to measure the approximation of a satisfied composition to a preference. This feature is implemented by a function that assigns a non-negative value to the preference bodies based on their evaluations.

Definition 7.2.11 (Approximation measurement function). For a given domain ontology $\langle \mathcal{L}, \Delta \rangle$, a composition CWS , and a preference body $pb = \langle f, sq \rangle$, the approximation function $Prx : PB \rightarrow \mathbb{Z}^+$ is defined as follows:

$$Prx(pb) = \begin{cases} |T| & \models_{T \cup C_{overall_q}} f \\ 0 & otherwise \end{cases}$$

where $T \subseteq \Delta$ and no $T' \subseteq \Delta$ exists such that $\models_{T' \cup C_{overall_q}} f$ and $|T'| < |T|$ \square

Based on the approximation function and evaluation of preferences, the total cost of a composition is calculated according to Definition 7.2.12.

Definition 7.2.12 (Composition total cost). The total cost of a composition CWS after evaluation of all the described preferences is

$TC_{CWS} = violations + approximations$, where

- $violations = \sum_i C_i$ where $CWS \not\models cf_i$ for $pref_i = \langle cf_i, C_i \rangle$
- $approximations = \sum_i \sum_{pb_j \in cf_i} \frac{Prx(pb_j)}{|\Delta|}$ such that $M_{CWS} \models cf_i$ for $pref_i = \langle cf_i, C_i \rangle$

\square

Note that the value of $\frac{Prx(pb_j)}{|\Delta|}$ is always less than or equal to one. The final definition is the ranker relation which is used to ranks the compositions based on their total cost.

Definition 7.2.13 (Ranker relation) Ranker relation \ll is a reflexive and transitive relation which ranks the compositions based on their total cost such that a composition with a lower cost has a better rank. Hence, $CWS_i \ll CWS_j$ means CWS_j has a better rank with lower cost in compared with CWS_i . \square

7.3 Evaluation

In order to evaluate our proposed approach, the example in Chapter 3 is formally modeled using our proposed framework. The four desired Web services are then described as the queries, and the framework attempts to answer them by composing the Web services in the assumed registry and ranks the results based on user preferences. We then investigate the ranked results to verify the correctness of our system.

7.3.1 Domain Ontology

First, the domain ontology must be described. As mentioned in definition 6.1.1, an ontology is a tuple $\langle \mathcal{L}, \Delta \rangle$ where $\mathcal{L} = \langle \mathbb{C}, \mathbb{V}, \mathbb{P} \rangle$, and Δ is the set of datalog rules in terms of language \mathcal{L} . \mathbb{C} is a set of constants which are names for our real objects in the domain of interest; for example, the name ‘*Toronto*’ can be used to refer to the real city Toronto, or the constant *AC120* can be used to refer to a real flight of AirCanada with this name. For simplicity, we assume that we have no constants (names) in our domain. The set \mathbb{V} is the set of all variable names that can be countably infinite. So, we register variables $a, a_1, a_2, \dots, b, b_1, b_2, \dots, c, c_1, c_2, \dots, z, z_1, z_2, \dots$, which are countably infinite, in the set \mathbb{V} .

The set \mathbb{P} is the set of all predicates. Defining the required predicates is the responsibility of domain experts and system engineers. The design of two experts may be completely different, but each design is expected to be expressive enough to capture all the needs of system users. Our registered predicates in \mathbb{P} and their definitions are described in the table 7.1.

Predicate	Description
-----------	-------------

Accommodation(x)	It verifies whether the given x is an accommodation
Hotel(x)	verifies whether the given x is a hotel
B_B(x)	verifies whether the given x is a B&B
Motel(x)	verifies whether the given x is a motel
3-starHotel(x)	verifies whether the given x is a 3 star hotel
4-starHotel(x)	verifies whether the given x is a 4 star hotel
5-starHotel(x)	verifies whether the given x is a 5 star hotel
LocatedIn(x, y)	verifies whether the given x is located in y
Trip(x, y)	verifies whether it is possible to move from x to y
Flight(x,y)	verifies whether there is a flight from x to y
Train(x,y)	verifies whether there is a train from x to y
World(x)	verifies whether the given x is a location in the world
NA(x)	verifies whether the given x is located in North America.
AS(x)	verifies whether the given x is located in Asia
AF(x)	verifies whether the given x is located in Africa
EU(x)	verifies whether the given x is located in Europe
Scandinavia(x)	verifies whether the given x is located in Scandinavia
Sweden(x)	verifies whether the given x is located in Sweden
\perp	implies bottom (nothing)
\top	implies top (everything)

Table 7.1: Registered Predicates in domain ontology

After specifying the language \mathcal{L} , the set of ontology rules can be expressed as follows:

Table 7.2: Domain ontology rules

(1) $Hotel(x) \rightarrow Accommodation(x)$	(11) $AS(x) \rightarrow World(x)$
(2) $B_B(x) \rightarrow Accommodation(x)$	(12) $EU(x) \rightarrow World(x)$
(3) $Motel(x) \rightarrow Accommodation(x)$	(13) $Scandinavia(x) \rightarrow EU(x)$
(4) $3 - starHotel(x) \rightarrow Hotel(x)$	(14) $Sweden(x) \rightarrow Scandinavia(x)$
(5) $4 - starHotel(x) \rightarrow 3starHotel(x)$	(15) $NA(x) \wedge AF(x) \rightarrow \perp$
(6) $5 - starHotel(x) \rightarrow 4starHotel(x)$	(16) $NA(x) \wedge AS(x) \rightarrow \perp$
(7) $Flight(x, y) \rightarrow Trip(x, y)$	(17) $AF(x) \wedge AS(x) \rightarrow \perp$
(8) $Train(x, y) \rightarrow Trip(x, y)$	(18) $NA(x) \wedge EU(x) \rightarrow \perp$
(9) $NA(x) \rightarrow World(x)$	(19) $EU(x) \wedge AF(x) \rightarrow \perp$
(10) $AF(x) \rightarrow World(x)$	(20) $AS(x) \wedge EU(x) \rightarrow \perp$

The fifth rule, for example, expresses that all the 4-star hotels are also 3-star hotels, and the rules 15-20 indicate that the continents are distinct. According to the rules, the Δ -graph is depicted in figure 7.2.

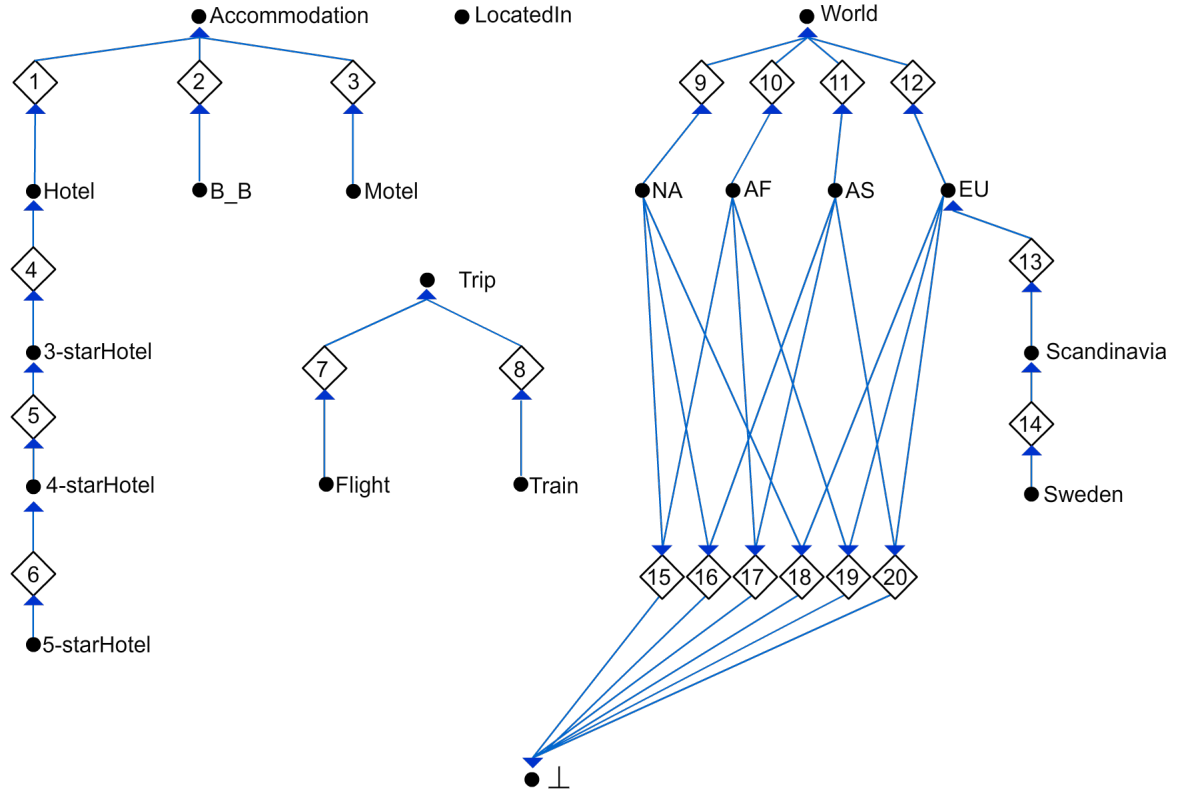


Figure 7.2: The Δ -graph of case study

7.3.2 Registered Web services

Since our focus is on DP services, the conjunctive query form can be used for service description. The descriptions (see definition 7.1.2) of registered Web services are represented in the following table:

Web service	Head	Description
E-Train	ET	$ET(\$a_1, a_2):- EU(a_1), EU(a_2), Train(a_1, a_2)$
Scandinavia-Train	ST	$ST(\$b_1, b_2):- Scandinavia(b_1), Scandinavia(b_2), Train(b_1, b_2)$
Europe-Flight	EF	$EF(\$c_1, c_2):- EU(c_1), EU(c_2), Flight(c_1, c_2)$
Local-Sweden Flight	LSF	$LSF(\$d_1, d_2):- Sweden(d_1), Sweden(d_2), Flight(d_1, d_2)$
WestJet	WJ	$WJ(\$e_1, e_2):- World(e_1), World(e_2), Flight(e_1, e_2)$
Egypt-Airlines	EGPF	$EGPF(\$m_1, m_2):- AF(m_1), AF(m_2), Flight(m_1, m_2)$
AirCanada	AC	$AC(\$n_1, n_2):- World(n_1), World(n_2), Flight(n_1, n_2)$
Star Hotel	SThtl	$SThtl(\$f_1, f_2):- World(f_1), 4\text{-starHotel}(f_2), LocatedIn(f_2, f_1)$
Affordable hotel	AFhtl	$AFhtl(\$g_1, g_2):- World(g_1), 3\text{-starHotel}(g_2), LocatedIn(g_2, g_1)$
Royal Hotel	SHhtl	$SHhtl(\$h_1, h_2):- World(h_1), 5\text{-starHotel}(h_2), LocatedIn(h_2, h_1)$

Table 7.3: Registered Web services in the service registry

7.3.3 Queries and Preferences

Each desired Web service in Chapter 3 can be described as a query, and associated preferences can be properly specified by our formal framework.

1) Simple Trip:

- $Q(\$x, y):- \text{World}(x), \text{Trip}(x, y), \text{World}(y)$
- Preferences= $\{\langle pb_1, 100 \rangle\}$ where:
 - $pb_1 = \langle \top \rightarrow \text{Flight}(x, y), \{\text{Trip}(x, y)\} \rangle$

2) Fast one-stop round trip travel:

- $Q(\$x, y):- \text{World}(x), \text{Trip}(x, u), \text{World}(u), \text{Trip}(u, y), \text{World}(y), \text{Trip}(y, w), \text{World}(w), \text{Trip}(w, x)$
- Preferences= $\{\langle pb_1, 100 \rangle, \langle pb_2, 100 \rangle, \langle pb_3, 400 \rangle, \langle pb_4, 400 \rangle\}$ where:
 - $pb_1 = \langle \top \rightarrow \text{Flight}(x, u), \{\text{Trip}(x, u)\} \rangle$
 - $pb_2 = \langle \top \rightarrow \text{Flight}(u, y), \{\text{Trip}(u, y)\} \rangle$
 - $pb_3 = \langle \top \rightarrow \text{Flight}(y, w), \{\text{Trip}(y, w)\} \rangle$
 - $pb_4 = \langle \top \rightarrow \text{Flight}(w, x), \{\text{Trip}(w, x)\} \rangle$

3) Visiting four European cities

- $Q(\$x, u, w, y):- \text{EU}(x), \text{Trip}(x, u), \text{EU}(u), \text{Trip}(u, w), \text{EU}(w), \text{Trip}(w, y), \text{EU}(y)$
- Preferences= $\{\langle (pb_1 \wedge pb_2) \rightarrow pTrain_1, 100 \rangle, \langle (pb_2 \wedge pb_3) \rightarrow pTrain_2, 100 \rangle, \langle (pb_2 \wedge pb_3) \rightarrow pTrain_3, 100 \rangle, \}$ where:

- $pb_1 = \langle \top \rightarrow Scandinavia(x), \{EU(x)\} \rangle$
- $pb_2 = \langle \top \rightarrow Scandinavia(u), \{EU(u)\} \rangle$
- $pb_3 = \langle \top \rightarrow Scandinavia(w), \{EU(w)\} \rangle$
- $pb_4 = \langle \top \rightarrow Scandinavia(y), \{EU(y)\} \rangle$
- $pTrain_1 = \langle \top \rightarrow Train(x, u), \{Trip(x, u)\} \rangle$
- $pTrain_2 = \langle \top \rightarrow Train(u, w), \{Trip(u, w)\} \rangle$
- $pTrain_3 = \langle \top \rightarrow Train(w, y), \{Trip(w, y)\} \rangle$

4) Trip around the world (visiting 5 stops):

- $Q(\$x_1, x_2, \dots, x_5) :- World(x_1), Trip(x_1, x_2), World(x_2), \dots, Trip(x_4, x_5), World(x_5), Trip(x_5, x_1)$
- Preferences = $\{ \langle 3 \preceq pb_1 \odot \dots \odot pb_5, 150 \rangle \}$ where:
 - $pb_1 = \langle \top \rightarrow AF(x_1), \{World(x_1)\} \rangle$
 - $pb_2 = \langle \top \rightarrow AF(x_2), \{World(x_2)\} \rangle$
 - $pb_3 = \langle \top \rightarrow AF(x_3), \{World(x_3)\} \rangle$
 - $pb_4 = \langle \top \rightarrow AF(x_4), \{World(x_4)\} \rangle$
 - $pb_5 = \langle \top \rightarrow AF(x_5), \{World(x_5)\} \rangle$

5) Affordable travel:

- $Q(\$x, y, h) :- World(x), Trip(x, y), World(y), LocatedIn(h, y), Hotel(h)$

- Preferences = $\{ \langle pb_1 \rightarrow pb_2, 100 \rangle \}$ where:
 - $pb_1 = \langle \top \rightarrow EU(y), \{World(y)\} \rangle$
 - $pb_2 = \langle \top \rightarrow 3 - starHotel(h), \{Hotel(h)\} \rangle$

7.3.4 Composition and Verification

In this Section, the composition process for the first and the fourth queries which are described in Section 7.3.3 are considered. Our proposed framework provides possible composition by rewriting queries in terms of registered Web services in the registry. To rewrite the query, MiniCon-FD⁺ first finds all the Δ -contained queries whose subgoals are appeared in at-least one view by finding CC-rules and expanding the initial query. It then rewrites each query by calling MiniCon and stores the results. Finally, the executability and composability of rewritten queries are checked, and those that are verified are returned as the results of the composition process.

To rewrite the first query, the following Δ -contained queries can be constructed with respect to the ontology as some of the possible Δ -contained quires:

- $Q_1(\$x, y):- World(x), Flight(x, y), World(y)$
- $Q_2(\$x, y):- World(x), Train(x, y), World(y)$
- $Q_2(\$x, y):- EU(x), Train(x, y), EU(y)$
- $Q_3(\$x, y):- Scandinavia(x), Flight(x, y), EU(y)$

In the next step, each query Q_i as well as Q are rewritten by MiniCon. Some of the results of this step are represented as follows:

- $Q'_1(\$x, y): WJ(\$x, y)$

- $Q'_2(\$x, y)$: $WJ(\$x, y), AC(\$x, y), SThtl(\$y, h)$
- $Q'_3(\$x, y)$: $ET(\$x, y)$
- $Q'_4(\$x, y)$: $ST(\$x, y), EGPF(\$x, y), WJ(\$x, y)$

Rewritten query Q'_2 provides the flights from the given city x to some cities in the world such that these flights are registered in the database of both AirCanada and WestJet, and there is also a Star hotel in the destination. Thus, they can be considered as some answers to our query. Q'_4 is not composable because if we expand the ST and $EGPF$ we will get $EU(x) \wedge AF(x)$ which is not possible based on the nineteenth rule in Δ .

For the second query, the following Δ -contained queries can be constructed w.r.t Δ :

- $Q_1(\$x, y)$:- $EU(x), Train(x, u), Sweden(u), Train(u, y), EU(y), Train(y, w), EU(w), Flight(w, x)$
- $Q_2(\$x, y)$:- $World(x), Flight(x, u), World(u), Flight(u, y), World(y), Flight(y, w), EU(w), Flight(w, x)$
- $Q_3(\$x, y)$:- $EU(x), Train(x, u), AF(u), Flight(u, y), EU(y), Train(y, w), EU(w), Train(w, x)$
- $Q_4(\$x, y)$:- $World(x), Flight(x, u), Sweden(u), Flight(u, y), Sweden(y), Train(y, w), EU(w), Train(w, x)$
- $Q_5(\$x, y)$:- $World(x), Train(x, u), EU(u), Train(u, y), Sweden(y), Train(y, w), EU(w), Train(w, x)$

In the next step, each query Q_i as well as Q are rewritten by MiniCon. Some of the results of this step are represented as follows:

- $Q'_1(\$x, y)$:- $ET(\$x, u), ST(\$u, y), ET(\$y, w), EF(\$w, x)$

- $Q'_2(\$x, y) \text{:- } AC(\$x, u), WJ(\$u, y), AC(\$y, w), EF(\$w, x)$
- $Q'_3(\$x, y) \text{:- } ET(\$x, u), EGPF(\$u, y), ET(\$y, w), ST(\$w, x)$
- $Q'_4(\$x, y) \text{:- } AC(\$x, u), LSF(\$u, y), ET(\$y, w), ET(\$w, x)$
- $Q'_5(\$x, y) \text{:- } SThl(\$x, k_2), ET(\$x, u), ST(\$u, y), ET(\$y, w), ET(\$w, \$x)$

Views ET and $EGPF$ in Q'_3 are impossible because we would have $EU(u) \wedge AF(u)$ by expanding these views while $EU(x) \wedge AF(x) \rightarrow \perp$ (the ETs outputs cannot be used as the input of EGPF). Therefore, after performing the verifications, the above results will be reduced to Q'_1, Q'_2, Q'_4 , and Q'_5 which constructs the following compositions:

- $CWS_1 = \langle \{EU(x) \rightarrow World(x), Train(x, u) \rightarrow Trip(x, u), Train(u, y) \rightarrow Trip(u, y), Train(y, w) \rightarrow Trip(y, w), Flight(w, x) \rightarrow Trip(w, x), Sweden(u) \rightarrow World(u), EU(y) \rightarrow World(y)\}, \{C_1, C_2, C_3, C_4\}, Q'_2, 0 \rangle$ where
 - $C_1 = \langle \{(a_1, a_1), (a_2, a_2)\}, ET(a_1, a_2), \{x \rightarrow a_1, u \rightarrow a_2\}, \{1, 2\} \rangle$
 - $C_2 = \langle \{(b_1, b_1), (b_2, b_2)\}, ST(b_1, b_2), \{u \rightarrow b_1, y \rightarrow b_2\}, \{3, 4\} \rangle$
 - $C_3 = \langle \{(a_1, a_1), (a_2, a_2)\}, ET(a_1, a_2), \{y \rightarrow a_1, w \rightarrow a_2\}, \{5, 6, 7\} \rangle$
 - $C_4 = \langle \{(c_1, c_1), (c_2, c_2)\}, EF(c_1, c_2), \{w \rightarrow c_1, x \rightarrow c_2\}, \{8\} \rangle$
- $CWS_2 = \langle \{Flight(x, u) \rightarrow Trip(x, u), Flight(u, y) \rightarrow Trip(u, y), Flight(y, w) \rightarrow Trip(y, w), Flight(w, x) \rightarrow Trip(w, x)\}, \{C_1, C_2, C_3, C_4\}, Q'_2, 0 \rangle$ where
 - $C_1 = \langle \{(n_1, n_1), (n_2, n_2)\}, AC(n_1, n_2), \{x \rightarrow n_1, u \rightarrow n_2\}, \{1, 2\} \rangle$
 - $C_2 = \langle \{(e_1, e_1), (e_2, e_2)\}, WJ(e_1, e_2), \{u \rightarrow e_1, y \rightarrow e_2\}, \{3, 4\} \rangle$
 - $C_3 = \langle \{(l_1, l_1), (l_2, l_2)\}, AC(l_1, l_2), \{y \rightarrow l_1, w \rightarrow l_2\}, \{5, 6\} \rangle$
 - $C_4 = \langle \{(c_1, c_1), (c_2, c_2)\}, EF(c_1, c_2), \{w \rightarrow c_1, x \rightarrow c_2\}, \{7, 8\} \rangle$
- $CWS_3 = \langle \{Flight(x, u) \rightarrow Trip(x, u), Flight(u, y) \rightarrow Trip(u, y), Train(y, w) \rightarrow Trip(y, w), Train(w, x) \rightarrow Trip(w, x), Sweden(u) \rightarrow World(u), Sweden(y) \rightarrow World(y), EU(w) \rightarrow World(w)\}, \{C_1, C_2, C_3, C_4\}, Q'_4, 0 \rangle$ where
 - $C_1 = \langle \{(n_1, n_1), (n_2, n_2)\}, AC(n_1, n_2), \{x \rightarrow n_1, u \rightarrow n_2\}, \{1, 2\} \rangle$

- $C_2 = \langle \{(d_1, d_1), (d_2, d_2)\}, LSF(d_1, d_2), \{u \rightarrow d_1, y \rightarrow d_2\}, \{3, 4, 5\} \rangle$
- $C_3 = \langle \{(l_1, l_1), (l_2, l_2)\}, ET(l_1, l_2), \{y \rightarrow l_1, w \rightarrow l_2\}, \{6, 7\} \rangle$
- $C_4 = \langle \{(a_1, a_1), (a_2, a_2)\}, ET(a_1, a_2), \{w \rightarrow a_1, x \rightarrow a_2\}, \{8\} \rangle$
- $CWS_4 = \langle \{Train(x, u) \rightarrow Trip(x, u), Train(u, y) \rightarrow Trip(u, y), Train(y, w) \rightarrow Trip(y, w), Train(w, x) \rightarrow Trip(w, x), ET(u) \rightarrow World(u), Sweden(y) \rightarrow World(y), EU(w) \rightarrow World(w)\}, \{C_1, C_2, C_3, C_4, C_5\}, Q'_5, 0 \rangle$ where
 - $C_1 = \langle \{(f_1, f_1), (f_2, f_2)\}, SThl(f_1, f_2), \{x \rightarrow f_1, k_2 \rightarrow f_2\}, \{1\} \rangle$
 - $C_2 = \langle \{(n_1, n_1), (n_2, n_2)\}, ET(n_1, n_2), \{x \rightarrow n_1, u \rightarrow n_2\}, \{2, 3\} \rangle$
 - $C_3 = \langle \{(d_1, d_1), (d_2, d_2)\}, ST(d_1, d_2), \{u \rightarrow d_1, y \rightarrow d_2\}, \{4, 5\} \rangle$
 - $C_4 = \langle \{(l_1, l_1), (l_2, l_2)\}, ET(l_1, l_2), \{y \rightarrow l_1, w \rightarrow l_2\}, \{6, 7\} \rangle$
 - $C_5 = \langle \{(a_1, a_1), (a_2, a_2)\}, ET(a_1, a_2), \{w \rightarrow a_1, x \rightarrow a_2\}, \{8\} \rangle$

Note that Q'_5 is executable and not impossible. This composition provides all the train itineraries from x such that x is located in Europe (i.e., it satisfies Q_5), and it also implies that there exists a hotel in x (more filtering on the values of x).

For the fifth query (Affordable travel), the following rewritten queries are provided as some of the results:

- $Q1(\$x, y, h):- EF(\$x, y), SThl(\$y, h)$
- $Q2(\$x, y, h):- AC(\$x, y), LSF(\$d_1, y), SThl(\$y, h)$
- $Q3(\$x, y, h):- EF(\$x, y), SThl(\$y, h_1), AFhtl(\$y, h)$
- $Q4(\$x, y, h):- EGPF(\$x, y), SHhtl(\$y, h)$
- $Q5(\$x, y, h):- EGPF(\$x, y), AFHtl(\$y, h)$

After performing two classes of verification, Q_2 will be eliminated due to the lack of executable order.

7.3.5 Ranking the Composed Web Services based on user Preferences

As the final step of our framework, the verified rewritings should be ranked based on the user preferences. In this Section, we rank the results of second and fifth query, and we will then compare the ranked results with our expected rankings in order to evaluate the correctness of the ranking system.

To rank the results for the second query, all the preferences should be evaluated for each composition. To commence the preference evaluation for the composed Web service CWS_1 , the $Coverall_{q_i}$ for each preference body $pb_i = \langle f_i, q_i \rangle$ $i = 1, \dots, 4$, defined in Section 7.3.3 for the second query, should be specified. According to the definition of the coverall set described in Definition 7.2.5, the coverall sets for CWS_1 are

- $Coverall_{q_1} = \{Train(x, u), Trip(x, u)\}$
- $Coverall_{q_2} = \{Train(u, y), Trip(u, y)\}$
- $Coverall_{q_3} = \{Train(y, w), Trip(y, w)\}$
- $Coverall_{q_4} = \{Flight(w, x), Trip(w, x)\}$

According to the semantics of preference bodies which are explained in Definition 7.2.7, pb_1 is true if for any interpretation I , $I \models_{\Delta \cup Coverall_{q_1}} (\top \rightarrow Flight(x, u))$. In other words, if there is an interpretation I such that $I \not\models_{\Delta \cup Coverall_{q_1}} (\top \rightarrow Flight(x, u))$, then pb_1 is false. To create such a I , assume a domain D in which all the cities are connected by train. Thus, $Train(x, u)^{I, Z} = true$, $Trip(x, u)^{I, Z} = true$, and $Flight(x, u)^{I, Z} = false$ for any variable assignment Z in I , resulting $pb_1 = false$. Likewise, pb_2 , pb_3 and pb_4 are evaluated as false. For pb_4 , because for any interpretation I , $I \models_{\Delta \cup \{Flight(w, x), Trip(w, x)\}} \top \rightarrow Flight(w, x)$ is true, pb_4 is true. Consequently,

Model M_{CWS_1} is $M_{CWS_1} = \{pb_1 = false, pb_2 = false, pb_3 = false, pb_4 = true\}$.

In the final step, the complex preference formulas of preferences should be investigated by using model M_{CWS_1} . The first three preferences are violated, so their associated costs should be added to the total cost (TC) of CWS_1 , resulting $TC = 600$. By following the same steps for the other compositions, the following results will be achieved:

Composition	Cost	Description
CWS_2	0	All the legs are flight
CWS_1	600	Only the last leg is flight
CWS_3	800	Only the first two legs are flight
CWS_4	1000	All the legs are train

As the above table shows, the composite Web services are ranked correctly as we expected based on their itineraries. Note that although CWS_3 provides two legs flight it has a lower rank compared to CWS_1 which has only one leg flight because the return legs are strongly preferred to be flight.

In order to rank the results for the fifth query, a similar procedure needs to be performed. Starting with $Q1(\$x, y, h) : -EF(\$x, y), SThtl(\$y, h)$, both preference bodies $pb_1 = \langle \top \rightarrow EU(y), \{World(y)\} \rangle$ and $pb_2 = \langle \top \rightarrow EU(y), \{World(y)\} \rangle$ are satisfied, thus the preference is satisfied. Likewise, $Q'2$ and $Q'3$ satisfy the preference. Queries $Q'4$ and $Q'5$ are also satisfied since the destination is not located in Europe. Consequently, no cost caused by violation of the preference is added to the compositions' total cost; however, the user prefers 3-star hotels. In this case the

proximate measurement function plays a role to order the compositions which are more close to our preference.

According to Definition 7.2.11, $Prox(pb_i)$ is equal to the minimum number of rules in Δ which is necessary to derive pb_i as true. For example, rules 4 and 5 in Δ are enough to make pb_2 as true, thus $Prox(pb_2) = 2$. The total costs (TCs) for the compositions according to Definition 7.2.12 are presented in the following table.

Composition	Proximation	Violation	TC	Description
CWS_1	$\frac{(0+2)}{21}$	0	$\frac{2}{21}$	destination: EU, Hotel: 4-star
CWS_2	$\frac{(2+2)}{21}$	0	$\frac{4}{21}$	destination: Sweden, Hotel: 4-star
CWS_3	$\frac{(0+0)}{21}$	0	0	destination: EU, Hotel: 3-star
CWS_4	$\frac{(0+3)}{21}$	0	$\frac{3}{21}$	destination: Africa, Hotel: 5-star
CWS_5	$\frac{(0+0)}{21}$	0	0	destination: Africa, Hotel: 3-star

As the above table shows, the compositions that provide flights to Europe and book a room in a 3-star hotel have a higher ranking. Moreover, those compositions that their provided flights' destinations are not located in Europe, but they book a room in a 3-star hotel have a higher ranking which doesn't have any conflict with the user preferences.

Chapter 8

Conclusions and Future Work

The main objectives of this research have been to compose DP Web services in the presence of domain ontology and to provide a framework in order to utilize user preferences to rank the results of a composition. In order to accomplish these goals, MiniCon algorithm, a state-of-the-art bucket-based algorithm, was extended to handle full dependency rules (Chapter 6). In the next step, the extended algorithm was adopted for the Web service composition problem (Chapter 7, Section 7.1). To let users describe their preferences about their desired Web services, a formal framework was designed to encode the users' preferences and to use them in order to rank the results of a composition (Chapter 7, Section 7.2). Finally, a running example, defined in Chapter 3, was implemented in order to evaluate the proposed framework (Chapter 7, Section 7.3). The primary contributions of this research as well as the potential future directions are outlined in the remainder of this Chapter.

8.1 Research Contributions

8.1.1 Query Rewriting in the Presence of Full Dependencies

While the Web facilitates fast access to various data sources distributed in different locations, finding the most complete answer to a given query in a reasonable time becomes a challenge. In order to make this process automatically, an additional ontological structure is required since data sources can be created anonymously with different terminologies. By utilizing more information about the domain, we may find more answers to a received query. Therefore, this ontological structure should also provide such information about the underlying domain.

A fundamental research question raised about providing such ontological structure was, *how can knowledge about a domain be encoded to be used during answering a query by integrating data sources?* To answer this research question, various logical structures have been reviewed which have been used to extend the data integration problem. Using the knowledge drawn from these studies, datalog rules, i.e. full dependencies, were selected as a language to build domain ontology because of its tractability and acceptable expressivity.

To utilize the ontology, the MiniCon algorithm was extended to integrate data sources in the presence of ontology domain. Nonetheless, MCDSAT and GQR algorithms (reviewed in Sections 4.4.3 and 4.4.4) are shown to be more efficient in comparison with MiniCon. The reason behind this decision is that since MiniCon provides information about which parts of a rewritten query are used to answer which parts of the initial query, making it suitable for handling the specific class of preferences called *local preferences*, introduced in Section 7.2.

To integrate data sources in the presence of full dependencies, MiniCon-FD and its optimized version, MiniCon-FD⁺, were proposed. These two algorithms were shown to utilize the ontology domain independent of their rewriting algorithm, i.e. MiniCon. Therefore, they can be used to extend GQR and MCDSAT as well. Moreover, in order to enhance the efficiency, MiniCon-FD⁺⁺ were proposed by modifying MiniCon to rewrite queries in the presence of full dependencies when no constant exists.

8.1.2 Preference-based Composition of DP Web Services

To compose DP Web Services, MiniCon-FD⁺ was chosen because of its capability of handling constant symbols. Because this algorithm was developed for data integration problem, it could not be used directly for Web service composition, thus an adaption were required. To verify the correctness of the results of MiniCon-FD⁺, two checking procedures are explained in Chapter 6 that test the executability and impossibility. Because thousands of composed Web services can be generated as the results of a composition process, finding the best fit composition based on user needs is a challenge. To assist users in the searching process, we utilized users' preferences about their desired Web services to rank the results. Our approach is motivated by the work proposed in [40], which provided a novel framework to utilize user preferences for ranking the results. However, since this approach used propositional logic as the language of preferences, some restrictions exist in describing preferences. In addition, to reveal the limitations of related works in encoding user preferences along with their issues in ranking, a running example was defined. This example was also used to evaluate our approach. To handle the shortcomings of related approaches, discussed in

Section 5.1, a novel formal framework was proposed.

In our proposed framework, two classes of *local* and *global preferences* can be described which the former encodes the user preferences about a fragment of query while the later encodes information about the entire query. In addition, a novel language is defined to capture user preferences. In order to rank the results of a composition accurately, the *proximate function* was defined to measure the closeness of a satisfied composite to the desired composite. To evaluate our framework, the running example was implemented.

8.2 Future Work

While the soundness of MiniCon-FD and its optimized version has been proved, the completeness of this algorithm still needs to be shown. Therefore, the first effort can be the completeness proof of this algorithm. We showed that MiniCon-FD can be added to any rewriting algorithm such as MCDSAT and GQR algorithms to extend them with full dependencies. However, we showed that some redundant information may be collected during rewriting process because we run MiniCon-FD separately. One way to enhance the performance is to break down these two algorithms and modify them.

On the other hand, improving the language of describing queries and Web services can enhance the efficiency and practability of system. MiniCon-FD⁺⁺ cannot support constant symbols but constants symbols are vital in practice for describing ontologies. Thus, one of our future efforts is to extend MiniCon-FD⁺⁺ with constant symbols. Moreover, using more expressive languages for describing ontology such as tgds rules

can be another option to improve the approach.

In our work, the user preferences regarding to functional characteristics of Web services are taken into account. However, considering the non-functional characteristics of Web services such as the price or response time can assist us in finding the most desirable ones. As such, incorporating a method to capture and use this type of preference in the composition process can be of value. In addition, user preferences can be used during the composition process to prune the cases that lead the system to undesirable composition. Nonetheless, this procedure can cause an empty result which may not be pleasant for users.

Bibliography

- [1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison Wesley, 1994.
- [2] S. Agarwal and S. Lamparter. User preference based automated selection of Web service compositions. In *ICSOC Workshop on Dynamic Web Processes*, pages 1–12, 2005.
- [3] Y. Arvelo, B. Bonet, and M. E. Vidal. Compilation of query-rewriting problems into tractable fragments of propositional logic. In *Proceedings of the 21st national conference on Artificial intelligence - Volume 1*, pages 225–230. AAAI Press, 2006.
- [4] F. Baader, S. Brand, and C. Lutz. Pushing the el envelope. In *International Joint Conference on Artificial Intelligence*, pages 364–369, 2005.
- [5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider., editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.

- [6] Q. Bai, J. Hong, and M. F. McTear. Query rewriting using views in the presence of inclusion dependencies. In *ACM international workshop on Web information and data management*, pages 134–138, 2003.
- [7] S. Bao, L. Zhang, C. Lin, and Y. Yu. A semantic rewriting approach to automatic information providing web service composition. In *Proceedings of the First Asian conference on The Semantic Web*, pages 488–500, 2006.
- [8] M. Barhamgi, D. Benslimane, and B. Medjahed. A query rewriting approach for web service composition. *Services Computing, IEEE Transactions on*, pages 1–10, 2010.
- [9] J. Barwise and J. Etchemendy. *Language, proof, and logic*. Seven bridges press, 1999.
- [10] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *International Colloquium on Automata, Languages and Programming*, pages 73–85. Springer Berlin Heidelberg, 1981.
- [11] K. Benouaret, D. Benslimane, A. HadjAli, and M. Barhamgi. Fudocs: A web service composition system based on fuzzy dominance for preference query answering. *PVLDB*, pages 1430–1433, 2011.
- [12] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.
- [13] B. Medjahed and A. Bouguettaya. *Service Composition for the Semantic Web*. Springer, 2011.

- [14] A. Cali, G. Gottlob, and A. Pieris. Towards more expressive ontology languages: The query answering problem. *Artificial Intelligence*, pages 87–128, 2012.
- [15] A. Cali and A. Pieris. On equality-generating dependencies in ontology querying - preliminary report. In *Alberto Mendelzon International Workshop on Foundations of Data Management*, pages 149–161, 2011.
- [16] D. Calvanese, G. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Automated Reasoning*, 39:385–429, 2007.
- [17] J. Cardoso. *Semantic Web Services: Theory, Tools and Applications*. IGI Global, 2007.
- [18] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 77–90, 1977.
- [19] L. Chang, F. Lin, and Z. Shi. A dynamic description logic for semantic web service. In *Semantics, Knowledge and Grid, Third International Conference on*, pages 74–79, 2007.
- [20] I. F. Cruz and H. Xiao. The role of ontologies in data integration. *JOURNAL OF ENGINEERING INTELLIGENT SYSTEMS*, pages 245–252, 2005.
- [21] F. Curbera, W. A. Nagy, and S. Weerawarana. Web services: Why and how. In *Object-Oriented Programming, Systems, Languages, and Applications - Workshop on Object-Oriented Web Services*, 2001.

- [22] A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11:11–34, 2000.
- [23] A. Darwiche. Decomposable negation normal form. *Journal of ACM*, 48(4):608–647, 2001.
- [24] O. Duschka and A. Y. Levy. Recursive plans for information gathering, 1997.
- [25] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *ACM SIGACT-SIGMOD-SIGART Conference on Principles of Database Systems*, pages 109–116, 1997.
- [26] O. M. Duschka and M. R. Genesereth. Query planning in infomaster. In *ACM Symposium on Applied Computing*, pages 109–111, 1997.
- [27] O. M. Duschka, M. R. Genesereth, and A. Y. Levy. Recursive query plans for data integration. *Journal of Logic Programming*, pages 49–73, 2000.
- [28] S. Dustdar and W. Schreiner. A survey on Web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.
- [29] T. Erl. *SOA Principles of Service Design*. Prentice Hall, 2008.
- [30] M. Ernst, T. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.
- [31] D. Fensel. *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*. Springer, 2003.

- [32] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. The vocabulary problem in human- system communication. *Communications of the ACM*, 30(11):964–971, 1987.
- [33] A. Gerevini and D. Long. Plan constraints and preferences in pddl3. Technical report, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005.
- [34] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.
- [35] J. Gryz. Query rewriting using views in the presence of functional and inclusion dependencies. *Information Systems*, pages 597 – 612, 1999.
- [36] A. Y. Halevy. Theory of answering queries using views. *SIGMOD*, pages 40–47, 2000.
- [37] P. Hitzler, M. Krötzsch, and S. Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.
- [38] I. Horrocks, O. Kutz, and U. Sattler. The even more irresistible sroiq. In *Knowledge Representation*, pages 57–67, 2006.
- [39] J. Hutchinson, G. Kotonya, J. Walkerdine, P. Sawyer, G. Dobson, and V. Onditi. The challenge of evolving existing systems to service-oriented architectures. In *IEEE International Conference on Industrial Informatics*, volume 2, pages 773–778, 2007.

- [40] D. Izquierdo, M.-E. Vidal, and B. Bonet. An expressive and efficient solution to the service selection problem. In *Proceedings of the semantic Web conference on The semantic web - Volume Part I*, pages 386–401, 2010.
- [41] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 164–169, 1982.
- [42] T. Kaczmarek and K. Wecel. Hype over service-oriented architecture continues. *Wirtschaftsinformatik*, 50:52–59, 2008.
- [43] S. Kambhampati. Refinement planning as a unifying framework for plan synthesis. *AI Magazine*, 18(2):67–97, 1997.
- [44] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop on Planning as Combinational Search*, pages 58–60, 1998.
- [45] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Joint Conference on Artificial Intelligence*, pages 318–325, 1999.
- [46] R. Klein, M. Buchheit, and W. Nutt. Configuration as model construction: The constructive problem solving approach. In *Artificial Intelligence in Design 94*, pages 201–218. 1994.

- [47] C. Koch. Query rewriting with symmetric constraints. In *Proceedings of the Second International Symposium on Foundations of Information and Knowledge Systems*, FoIKS '02, pages 130–147, 2002.
- [48] G. Konstantinidis and J. L. Ambite. Scalable query rewriting: a graph-based approach. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 97–108, 2011.
- [49] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 1999.
- [50] C. T. Kwok and D. S. Weld. Planning to gather information. In *In Proceedings of the AAAI Thirteenth National Conference on Artificial Intelligence*, pages 32–39, 1996.
- [51] U. D. Lago, M. Pistore, and P. Traverso. Planning with a language for extended goals. In *Eighteenth national conference on Artificial intelligence*, pages 447–454, 2002.
- [52] F. Lecue and N. Mehandjiev. Towards scalability of quality driven semantic Web service composition. In *IEEE International Conference on Web Services*, pages 469 –476, 2009.
- [53] M. Lenzerini. Logical foundations for data integration. pages 38–40, 2005.
- [54] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS, San Jose, Calif, USA*, pages 95 –104, 1995.

- [55] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *International Conference on Very Large Data Bases*, pages 251–262, 1996.
- [56] N. Lin, U. Kuter, and E. Sirin. Web service composition with user preferences. In *Proceedings of the European Semantic Web Conference*, pages 629–643, 2008.
- [57] N. Lin, U. Kuter, and E. Sirin. Web service composition with user preferences. In *Proceedings of the Semantic Web Conference on The semantic Web: Research and Applications*, pages 629–643, 2008.
- [58] J. Lu, Y. Yu, and J. Mylopoulos. A lightweight approach to semantic web service synthesis. In *Workshop on Challenges in Web Information Retrieval and Integration*, pages 240–247, 2005.
- [59] A. Maedche and S. Staab. Ontology learning for the semantic Web. *IEEE Intelligent Systems*, 16(2):72 – 79, 2001.
- [60] D. Maier and A. Mendelzon. Testing implications of data dependencies. *ACM Transactions on Database Systems*, pages 455–469, 1979.
- [61] P. T. Mallik Ghallab, Dana Nau. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.
- [62] S. Mcilraith and R. Fadel. Planning with complex actions. In *ProcProceedings of the International Workshop on Non-Monotonic Reasoning*, pages 356–364, 2002.

- [63] S. A. McIlraith and T. C. Son. Adapting Golog for composition of semanticWeb services. In *Principles of Knowledge Representation and Reasoning*, pages 482–496, 2002.
- [64] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems*, 16(2):46–53, 2001.
- [65] A. Mesmoudi, M. Mrissa, and M. Hacid. Combining configuration and query rewriting for web service composition. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 113 –120, 2011.
- [66] P. Mitra. An algorithm for answering queries efficiently using views. In *Australasian Database Conference*, pages 99–106, 2001.
- [67] W. Niu, Z. Shi, and L. Chang. *A Context Model for Service Composition Based on Dynamic Description Logic*. 2009.
- [68] OASIS. Reference model for service oriented architecture 1.0. @ONLINE, 2010.
- [69] S.-C. Oh, D. Lee, and S. R. T. Kumara. A comparative illustration of ai planning-based web services composition. *SIGecom Exch.*, 5:1–10, 2006.
- [70] S.-C. Oh, D. Lee, and S. R. T. Kumara. Web service planner (wspr): An effective and scalable web service composition algorithm. *International Journal of Web Services Research*, 4(1):1–22, 2007.
- [71] M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

- [72] P. Papazoglou, M. *Web Services: Principles and Technologies*. Prentice Hall, 2008.
- [73] J. Peer. Web service composition as AI planning-a survey. *Technical report, University of St. Gallen, Switzerland*, 2005.
- [74] R. Pottinger and A. Halevy. MiniCon: A scalable algorithm for answering queries using views. *The VLDB Journal*, 10(2-3):182–198, 2001.
- [75] H. Rahmani, G. GhasemSani, and H. Abolhassani. Automatic Web service composition considering user non-functional preferences. In *Proceedings of the International Conference on Next Generation Web Services Practices*, pages 33–38, 2008.
- [76] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 105–112, 1995.
- [77] J. Rao and X. Su. A survey of automated Web service composition methods. In *In Proceedings of the International Workshop on Semantic Web Services and Web Process Composition*, pages 43–54, 2004.
- [78] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

- [79] G. R. Santhanam, S. Basu, and V. Honavar. On utilizing qualitative preferences in Web service composition: A CP-net based approach. In *Proceedings of the IEEE Congress on Services - Part I*, pages 538–544, 2008.
- [80] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for Web service composition using shop2. *Web Semantics*, 1(4):377–396, 2004.
- [81] S. Sohrabi, J. A. Baier, and S. A. McIlraith. HTN planning with preferences. In *Proceedings of the Jont Conference on Artificial intelligence*, pages 1790–1797, 2009.
- [82] S. Sohrabi and S. A. Mcilraith. Optimizing Web service composition while enforcing regulations. In *Proceedings of the International Semantic Web Conference*, pages 601–617, 2009.
- [83] S. Sohrabi and S. A. Mcilraith. Optimizing Web service composition while enforcing regulations. In *Proceedings of the Semantic Web Conference*, pages 601–617, 2009.
- [84] S. Sohrabi, N. Prokoshyna, and S. A. Mcilraith. Web service composition via the customization of golog programs with user preferences. chapter Conceptual Modeling: Foundations and Applications, pages 319–334. 2009.
- [85] P. Traverso and M. Pistore. Automated composition of semantic web services into executable processes. pages 380–394, 2004.
- [86] G. Weikum. Letter from the special issue editor. *IEEE Data Eng. Bull.*, 25(1):3, 2002.

- [87] L. Zhou, H. Chen, T. Yu, J. Ma, and Z. Wu. Ontology-based scientific data service composition: A query rewriting-based approach. In *AAAI Spring Symposium: Semantic Scientific Knowledge Integration*, pages 116–121, 2008.